THOMAS H.CORMEN
CHARLES E.LEISERSON
RONALD R.RIVEST

INTRODUCERE ÎN ALGORITMI

Titlul original: ***Introduction to Algorithms***

Editor coordonator: Clara Ionescu

Coordonator traducere: Horia F. Pop

Traducere:

Prefaţa: Simona Motogna	Capitolul 19: Florian M. Boian
Capitolul 1: Horia F. Pop	Capitolul 20: Ioan Lazar
Capitolul 2: Paul Blaga	Capitolul 21: Ioan Lazar
Capitolul 3: Paul Blaga	Capitolul 22: Virginia Niculescu
Capitolul 4: Paul Blaga	Capitolul 23: Adrian Monea
Capitolul 5: Adrian Monea	Capitolul 24: Adrian Monea
Capitolul 6: Radu Trâmbiţaş	Capitolul 25: Luminiţa State
Capitolul 7: Clara Ionescu	Capitolul 26: Mihai Scorţaru
Capitolul 8: Zoltán Kása	Capitolul 27: Zoltán Kása, Simona Motogna
Capitolul 9: Luminiţa State	Capitolul 28: Zoltán Kása
Capitolul 10: Luminiţa State	Capitolul 29: Florian M. Boian
Capitolul 11: Simona Motogna	Capitolul 30: Mihai Scorţaru
Capitolul 12: Simona Motogna	Capitolul 31: Liviu Negrescu
Capitolul 13: Bazil Pârv	Capitolul 32: Radu Trâmbiţaş
Capitolul 14: Bazil Pârv	Capitolul 33: Liviu Negrescu
Capitolul 15: Bazil Pârv	Capitolul 34: Liana Bozga
Capitolul 16: Cristina Vertan	Capitolul 35: Liana Bozga
Capitolul 17: Cristina Vertan	Capitolul 36: Mihai Scorţaru
Capitolul 18: Cristina Vertan	Capitolul 37: Horia F. Pop

Lecturare:

Florian M. Boian, Liana Bozga, Carmen Bucur, Ioana Chiorean, Horia Georgescu, Clara Ionescu, Eugen Ionescu, Zoltán Kása, Ioan Lazar, Adrian Monea, Simona Motogna, Virginia Niculescu, Bazil Pârv, Horia F. Pop, Mihai Scorţaru, Radu Trâmbiţaş

Index: Simona Motogna

Grafica: Dan Creţu

Coperta: Mircea Drăgoi

Au confruntat cu originalul:

Carmen Bucur, Clara Ionescu, Simona Motogna, Dragoş Petraşcu

Filolog: cerc. princ. Ileana Câmpean

ISBN 973-97534-7-7

Cuprins

Prefaţa ediţiei în limba română

Stimaţi cititori,

Ani de zile am visat apariţia acestei cărţi. Ştiu că mulţi profesori, elevi şi studenţi au căutat lucrarea semnată de Thomas H. Cormen, Charles E. Leiserson şi Ronald L. Rivest în original sau diverse traduceri ale ei. Unii, mai norocoşi, au reuşit poate să împrumute cartea din diverse locuri, pentru scurte perioade de timp. Dar această carte conţine un volum foarte mare de cunoştinţe şi nu este suficient să o răsfoieşti într-un sfârşit de săptămână, ci este nevoie de luni, poate ani, pentru a putea asimila toate informaţiile cuprinse într-o lucrare ca aceasta.

Primul contact cu MIT Press l-am stabilit în toamna anului 1998. Mulţumim doamnei Cristina Sanmartin (MIT Press) că ne-a încurajat şi a intermediat semnarea contractului pentru traducere şi editare în limba română. Acum suntem deja în 2000. A durat traducerea, tehnoredactarea şi tipărirea. Pentru apariţia în limba română a cărţii, adresez mulţumiri profesorilor de la Universitatea "Babeş-Bolyai" şi de la Universitatea Bucureşti, studenţilor şi prietenilor care au sacrificat multe ore din timpul lor liber, pentru ca dumneavoastră să puteţi ţine această carte în mână.

Domnul conferenţiar dr. Horia F. Pop, de la Universitatea "Babeş-Bolyai", Facultatea de Matematică şi Informatică, a coordonat traducerea, a stabilit împreună cu traducătorii modul de traducere a termenilor de specialitate noi, începând cu index-ul, pentru a asigura o traducere coerentă, omogenă şi pe cât posibil unitară. De asemenea, a pregătit şi coordonat utilizarea de către traducători a sistemului LaTeX.

Am lucrat cu cadre didactice universitare de specialitate deoarece o asemenea carte nu se traduce doar cu lingvişti, este nevoie de specialişti din domeniu care înţeleg foarte bine conţinutul ştiinţific şi sunt de acord să reformuleze anumite părţi pentru a exprima ceea ce de fapt autorul a dorit să comunice. Mulţumim domnilor (în ordine alfabetică) lector dr. Paul Blaga, profesor dr. Florian Mircea Boian, drd. Liana Bozga, profesor dr. Zoltán Kása, lector drd. Ioan Lazar, lector drd. Simona Motogna, asistent drd. Virginia Niculescu, profesor dr. Bazil Pârv, lector dr. Radu Trâmbiţaş, de la Universitatea "Babeş-Bolyai" din Cluj, domnului profesor Liviu Negrescu de la Universitatea Tehnică din Cluj, domnului profesor dr. Horia Georgescu de la Academia de Studii Economice, doamnei profesor dr. Luminiţa State de la Universitatea din Piteşti, doamnei lector drd. Cristina Vertan, de la Universitatea Bucureşti şi nu în ultimul rând studenţilor Adrian Monea şi Mihai Scorţaru, de la Universitatea Tehnică din Cluj.

Cum aceasta este prima carte editată de Computer Libris Agora în LaTeX, tehnoredactarea a constituit şi ea un efort. În special pregătirea desenelor a ridicat multe probleme, graficianul

nostru Dan Creţu lucrând nenumărate ore la realizarea acestora. "Asistenţa tehnică" în domeniul pregătirii pentru tipar a fost acordată de Mihai Uricaru.

Într-o carte de asemenea dimensiuni şi complexitate cu siguranţă se strecoară şi unele greşeli. Am dori ca acestea să fie eliminate din viitoarele ediţii ale lucrării. Din acest motiv, vă rugăm să ne transmiteţi observaţiile şi sugestiile dumneavoastră pe adresa de e-mail:

`clara@cs.ubbcluj.ro`

sau pe adresa editurii:

Editura Computer Libris Agora
Str. Universităţii nr. 8
3400 Cluj Napoca
Tel.: 064-192422

Urmând exemplul american, lista cu greşelile găsite va fi disponibilă pe Internet. Cei care doresc să intre în posesia ultimei erate pot trimite un mesaj la adresa `algoritmi@agora.ro` conţinând expresia "erata algoritmi" în câmpul `Subject` şi vor primi automat ultima versiune în format electronic. Alternativ, puteţi vizita pagina

`http://www.libris.agora.ro/algoritmi/erata.html`.

Autorii, aşa cum am aflat de la MIT Press, lucrează la o versiune nouă a cărţii, revizuită şi îmbogăţită. Vă promitem că, imediat ce această versiune va fi disponibilă pentru traducere, o vom publica şi în limba română.

Cluj, 28 martie 2000

Clara Ionescu, editor coordonator

Prefaţă

Această carte se doreşte a fi o introducere exhaustivă în studiul modern al algoritmilor. Ea prezintă mulţi algoritmi şi îi studiază în profunzime, păstrând totuşi analiza şi proiectarea lor la un nivel accesibil tuturor cititorilor. Am încercat să păstrăm explicaţiile la un nivel elementar fără a afecta profunzimea abordării sau rigoarea matematică.

Fiecare capitol prezintă un algoritm, o tehnică de proiectare, o arie de aplicare sau un subiect legat de acestea. Algoritmii sunt descrişi în engleză şi în "pseudocod, care a fost astfel conceput încât să poată fi înţeles de oricine care posedă cunoştiinţe elementare de programare. Cartea conţine peste 260 de figuri, care ilustrează modul de funcţionare a algoritmilor. Deoarece considerăm *eficienţa* ca un criteriu de bază în proiectare, vom include analiza atentă a timpilor de execuţie a algoritmilor prezentaţi.

Cartea se adresează, în primul rând, celor care studiază un curs elementar sau academic de algoritmi sau structuri de date. Deoarece studiază atât aspecte tehnice, cât şi matematice în proiectarea algoritmilor, este la fel de binevenită pentru profesionişti autodidacţi.

Pentru profesori

Această carte a fost concepută pentru a fi atât multilaterală cât şi completă. O veţi găsi utilă pentru diverse cursuri, de la un curs elementar de structuri de date până la un curs academic de algoritmi. Deoarece am adunat considerabil mai mult material decât poate "încăpea" într-un curs obişnuit de un semestru, cartea poate fi privită ca un "depozit" din care se poate alege materialul cel mai adecvat cursului care urmează să fie prezentat.

Vi se va părea uşor să vă organizaţi cursul doar pe baza capitolelor care vă sunt necesare. Am conceput capitolele relativ de sine stătătoare, astfel încât să nu depindeţi în mod neaşteptat sau inutil de informaţii din alte capitole. Fiecare capitol prezintă la început noţiunile mai simple şi apoi, cele mai dificile, partiţionarea pe secţiuni mărginind în mod natural punctele de oprire. Pentru un curs elementar se pot folosi doar primele secţiuni dintr-un capitol; pentru un curs avansat se poate parcurge întreg capitolul.

Am inclus peste 900 de exerciţii şi peste 120 de probleme. Fiecare secţiune se încheie cu exerciţii, iar fiecare capitol cu probleme. Exerciţiile sunt, în general, întrebări scurte care testează stăpânirea noţiunilor de bază prezentate. Unele dintre ele sunt simple verificări, în timp ce altele sunt potrivite ca teme de casă. Problemele reprezintă studii de caz mult mai elaborate, care de multe ori introduc noţiuni noi; în general, constau din mai multe întrebări care conduc studentul prin paşii necesari pentru a ajunge la o soluţie.

Am notat cu (*) secţiunile şi exerciţiile care se adresează mai degrabă studenţilor avansaţi decât celor începători. O secţiune marcată cu (*) nu este în mod necesar mai dificilă decât

una obişnuită, dar poate necesita înţelegerea unor noţiuni matematice mai complexe. În mod asemănător, exerciţiile marcate cu (*) pot necesita cunoştinţe mai avansate sau creativitate mai mult decât medie.

Pentru studenţi

Sperăm ca textul acestei cărţi să ofere o introducere plăcută în domeniul algoritmilor. Am intenţionat să prezentăm fiecare algoritm într-o manieră accesibilă şi interesantă. Pentru a fi un ajutor când întâlniţi algoritmi neobişnuiţi sau dificili, am descris fiecare algoritm pas cu pas. De asemenea, am oferit explicaţii detaliate a noţiunilor matematice necesare înţelegerii analizei algoritmilor. Dacă deja dispuneţi de cunoştinţe relative la un anumit subiect, atunci veţi găsi capitolele organizate astfel încât puteţi frunzări secţiunile introductive şi să vă concentraţi asupra noţiunilor mai complexe.

Fiind o carte vastă, e posibil ca ceea ce vi se predă să acopere doar o parte a acestui material. Oricum, ne-am străduit să construim această carte utilă şi ca suport de curs, şi ca o referinţă teoretică şi pratică în cariera voastră viitoare.

Care sunt cerinţele pentru a înţelege această carte?

- Este necesară o anumită experienţă de programare. În particular, trebuie să înţelegeţi procedurile recursive şi structurile de date simple, ca tablouri sau liste înlănţuite.
- Este necesară o anumită experienţă în demonstraţii prin inducţie matematică. Unele porţiuni din carte se bazează pe anumite cunoştinţe de calcul elementar. În afară de asta, partea întâi a acestei cărţi vă prezintă toate tehnicile matematice necesare.

Pentru profesionişti

Aria largă de subiecte prezentate în această carte o recomandă ca un excelent manual de algoritmi. Deoarece fiecare capitol este conceput relativ independent, vă puteţi concentra doar asupra subiectelor care vă interesează.

Majoritatea algoritmilor luaţi în discuţie au o mare aplicabilitate practică. Din acest motiv, luăm în studiu şi aspecte legate de implementare şi alte probleme "inginereşti". În general, am propus alternative practice puţinilor algoritmi care prezintă interes primordial teoretic.

Dacă doriţi să implementaţi oricare dintre algoritmi, veţi constata că transcrierea algortimilor din pseudocod în limbajul de programare preferat este o sarcină relativ uşoară. Limbajul pseudocod este astfel proiectat încât să prezinte fiecare algoritm într-un stil clar şi concis. În consecinţă, nu luăm în considerare tratarea erorilor sau alte aspecte tehnice care necesită presupuneri specifice relative unor medii de programare. Am urmărit să prezentăm fiecare algoritm simplu şi direct, fără a ne opri asupra detaliilor relative la limbaje de programare.

Erori

O carte de asemenea dimensiune este în mod cert supusă erorilor şi omisiunilor. Dacă descoperiţi o eroare sau aveţi unele sugestii constructive, am aprecia să ni le împărtăşiţi. Sunt binevenite, în mod special, sugestii pentru noi exerciţii şi probleme, dar vă rugăm să includeţi şi soluţiile. Le puteţi trimite prin poştă, pe adresa:

Introduction to Algorithms
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139

Sau, puteţi folosi poşta electronică pentru a semnala erori, pentru a cere o erată sau pentru a face sugestii constructive. Pentru a primi instrucţiunile de rigoare, trimiteţi un mesaj pe adresa `algorithms@theory.lcs.mit.edu` cu **`"Subject: help"`** în antetul mesajului. Regretăm că nu putem răspunde personal la toate mesajele.

Mulţumiri

Mai mulţi prieteni şi colegi au contribuit la calitatea acestei cărţi. Le mulţumim tuturor pentru ajutorul dat şi pentru observaţiile constructive pe care ni le-au adresat.

Laboratorul de Informatică al Universităţii MIT a reprezentat un cadru de lucru ideal. În mod deosebit, colegii noştri de la laboratorul Grupului de Teoria Calculatoarelor ne-au ajutat şi acceptat cererile noastre continue de parcurgere critică a capitolelor. Dorim să mulţumim în mod special următorilor colegi: Baruch Awerbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David Shmoys şi Eva Tardos. Mulţumiri lui William Ang, Sally Bemus, Ray Hirschfeld şi Mark Reinhold pentru întreţinerea calculatoarelor (DEC Microvax, Apple Macintosh şi Sun Sparcstation) şi pentru recompilarea TEX ori de câte ori am depăşit limita timpului de compilare. Menţionăm ajutorul oferit lui Charles Leiserson de Machines Corporation, pentru a putea lucra la această carte chiar şi în perioada absenţei sale de la MIT.

Mulţi dintre colegii noştri au folosit manuscrise ale acestei cărţi pentru cursuri predate la alte şcoli, sugerând numeroase corecţii şi revizuiri. Dorim să mulţumim în mod deosebit următorilor: Richard Beigel (Yale), Andrew Goldberg (Stanford), Joan Lucas (Rutgers), Mark Overmars (Utrecht), Alan Sherman (Tufts şi Maryland) şi Diane Souvaine (Rutgers).

De asemenea, mulţi asistenţi ai cursurilor noastre au avut contribuţii majore la dezvoltarea acestei cărţi. Mulţumim în mod special următorilor: Alan Baratz, Bonnie Berger, Aditi Dhagat, Burt Kaliski, Arthur Lent, Andrew Moulton, Marios Papaefthymiou, Cindy Phillips, Mark Reinhold, Phil Rogaway, Flavio Rose, Arie Rudich, Alan Sherman, Cliff Stein, Susmita Sur, Gregory Troxel şi Margaret Tuttle.

Asistenţa tehnică necesară a fost asigurată de mai multe persoane. Denise Sergent a petrecut multe ore în bibliotecile universităţii MIT pentru a căuta referinţe bibliografice. Maria Sensale, bibliotecara sălii noastre de lectură, a fost tot timpul deosebit de săritoare. Accesul la biblioteca personală a lui Albert Meyer ne-a economisit mult timp în pregătirea referinţelor bibliografice. Shlomo Kipnis, Bill Niehaus şi David Wilson au demonstrat vechi exerciţii, au propus unele noi şi au schiţat notiţe relative la rezolvarea lor. Marios Papaefthymiou şi Gregory Troxel au contribuit la realizarea indexului. De-a lungul anilor, secretarele nostre: Inna Radzihovsky, Denise Sergent, Gayle Sherman şi în mod deosebit Be Hubbard au fost un sprijin continuu în acest proiect, pentru care le aducem mulţumiri.

Multe erori din primele manuscrise au fost detectate de studenţi. Mulţumim în mod special următorilor: Bobby Blumofe, Bonnie Eisenberg, Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge, John Pezaris, Steve Ponzio şi Margaret Tuttle pentru parcurgerile şi corecturile efectuate.

De asemenea, colegii noştri au adus critici asupra anumitor capitole, ne-au oferit indicaţii asupra unor algoritmi specifici, motiv pentru care le suntem îndatoraţi. Mulţumim în mod

deosebit următorilor: Bill Aiello, Alok Aggarwal, Eric Bach, Vašek Chvátal, Richard Cole, Johan Hastad, Alex Ishii, David Johnson, Joe Killian, Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plambeck, Hershel Safer, Jeff Shallit, Cliff Stein, Gil Strang, Bob Tarjan şi Paul Wang. Câţiva colegi ne-au oferit probleme propuse; le mulţumim următorilor: Andrew Goldberg, Danny Sleator şi Umesh Vazirani.

Această carte a fost redactată cu LaTeX, un pachet de macrouri TeX. Figurile au fost desenate pe Apple Macintosh folosind MacDraw II; mulţumim lui Joanna Terry de la Claris Corporation şi lui Michael Mahoney de la Advanced Computer Graphics pentru sprijinul lor. Indexul a fost compilat folosind Windex, un program C scris de autori. Bibliografia a fost pregătită folosind BibTeX. Cartea a fost tipărită la American Mathematical Society cu ajutorul unei maşini Autologic; mulţumim lui Ralph Youngen de la AMS pentru ajutorul dat. Coperta cărţii a fost concepută de Jeannet Leendertse. Design-ul cărţii a fost creat de Rebecca Daw, iar Amy Hendrickson l-a implementat în LaTeX.

Colaborarea cu MIT Press şi McGraw-Hill a fost deosebit de plăcută. Mulţumim în mod special următorilor: Frank Satlow, Terry Ehling, Larry Cohen şi Lorrie Lejeune de la MIT Press şi David Shapiro de la McGraw-Hill pentru încurajări, sprijin şi răbdare. Suntem deosebit de recunoscători lui Larry Cohen pentru editarea excepţională.

În fine, mulţumim soţiilor noastre – Nicole Cormen, Linda Lue Leiserson şi Gail Rivest – şi copiilor noştri – Ricky, William şi Debby Leiserson şi Alex şi Christopher Rivest – pentru dragostea şi sprijinul lor pe întreaga durată a scrierii acestei cărţi. (Alex Rivest ne-a ajutat în mod special cu "Paradoxul zilei de naştere a marţianului"). Dragostea, răbdarea şi încurajarea din partea familiilor noastre au făcut acest proiect posibil. Le dedicăm din suflet această carte lor.

Cambridge, Massachusetts
Martie, 1990

Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest

1 Introducere

Acest capitol vă va familiariza cu conceptele de bază folosite în această carte, referitor la proiectarea şi analiza algoritmilor. Este un text, practic, independent de restul cărţii, dar include câteva referiri la materialul care va fi introdus în partea I.

Începem cu o discuţie asupra problemelor generale ale calculabilităţii şi ale algoritmilor necesari pentru rezolvarea acestora, cu problema sortării, ca exemplu introductiv. Pentru a arăta cum vom specifica algoritmii prezentaţi, vom introduce un "pseudocod", care ar trebui să fie familiar cititorilor obişnuiţi cu programarea. Sortarea prin inserţie, un algoritm simplu de sortare, va servi ca exemplu iniţial. Vom analiza timpul de execuţie pentru sortarea prin inserţie, introducând o notaţie care să descrie modul în care creşte acest timp o dată cu numărul obiectelor aflate în operaţia de sortare. De asemenea, vom introduce în proiectarea algoritmilor metoda *divide şi stăpâneşte*, pe care o vom utiliza pentru dezvoltarea unui algoritm numit sortare prin interclasare. Vom încheia cu o comparaţie între cei doi algoritmi de sortare.

1.1. Algoritmi

Fără a fi foarte exacţi, spunem că un ***algoritm*** este orice procedură de calcul bine definită care primeşte o anumită valoare sau o mulţime de valori ca ***date de intrare*** şi produce o anumită valoare sau mulţime de valori ca ***date de ieşire***. Astfel, un algoritm este un şir de paşi care transformă datele de intrare în date de ieşire.

Putem, de asemenea, să privim un algoritm ca pe un instrument de rezolvare a ***problemelor de calcul*** bine definite. Enunţul problemei specifică, în termeni generali, relaţia dorită intrare/ieşire. Algoritmul descrie o anumită procedură de calcul pentru a se ajunge la această legătură intrare/ieşire.

Vom începe studiul algoritmilor cu problema sortării unui şir de numere în ordine nedescrescătoare. Această problemă apare frecvent în practică şi furnizează o bază foarte utilă pentru introducerea multor metode standard pentru proiectarea şi analiza algoritmilor. Iată cum vom defini formal ***problema sortării***:

Date de intrare: Un şir de n numere $\langle a_1, a_2, \ldots, a_n \rangle$.

Date de ieşire: O permutare (reordonare) a şirului iniţial, $\langle a'_1, a'_2, \ldots, a'_n \rangle$ astfel încât $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Fiind dat un şir de intrare, ca, de exemplu, $\langle 31, 41, 59, 26, 41, 58 \rangle$, un algoritm de sortare returnează, la ieşire, şirul $\langle 26, 31, 41, 41, 58, 59 \rangle$. Un şir de intrare ca cel de mai sus se numeşte o ***instanţă*** a problemei de sortare. În general, prin ***instanţă a unei probleme*** se va înţelege mulţimea tuturor datelor de intrare (care satisface restricţiile impuse în definirea problemei) necesare pentru a determina o soluţie a problemei.

Sortarea este o operaţie fundamentală în informatică (multe programe o folosesc ca pas intermediar) şi, ca urmare, a fost dezvoltat un număr mare de algoritmi de sortare. Care algoritm este cel mai bun pentru o aplicaţie dată depinde de numărul de obiecte care trebuie sortate, de

Figura 1.1 Modul de sortare a unei mâini de cărţi, utilizând sortarea prin inserţie.

gradul în care aceste obiecte sunt deja sortate într-un anumit fel şi de tipul de mediu electronic care urmează să fie folosit: memorie principală, discuri sau benzi magnetice.

Un algoritm este ***corect*** dacă, pentru orice instanţă a sa, se termină furnizând ieşirea corectă. Vom spune că un algoritm corect ***rezolvă*** problema de calcul dată. Un algoritm incorect s-ar putea să nu se termine deloc în cazul unor anumite instanţe de intrare, sau s-ar putea termina producând un alt răspuns decât cel dorit. Contrar a ceea ce s-ar putea crede, algoritmii incorecţi pot fi uneori utili dacă rata lor de eroare poate fi controlată. Vom vedea un exemplu în capitolul 33, când vom studia algoritmi pentru găsirea numerelor prime foarte mari. Totuşi, în general ne vom ocupa doar de algoritmi corecţi.

Concret, un algoritm poate fi specificat printr-un program pentru calculator sau chiar ca un echipament hardware. Singura condiţie este aceea ca specificaţiile să producă o descriere precisă a procedurii de calcul care urmează a fi parcursă.

În această carte vom descrie algoritmii sub forma unor programe scrise într-un ***pseudocod*** care seamănă foarte mult cu limbajele C, Pascal sau Algol. Dacă sunteţi cât de cât familiarizaţi cu oricare dintre acestea, nu veţi avea probleme în a citi algoritmii noştri. Ceea ce diferenţiază codul "real" de pseudocod este faptul că vom folosi metoda cea mai clară şi mai concisă pentru a descrie un algoritm dat. O altă diferenţă dintre pseudocod şi codul real este aceea că pseudocodul nu se ocupă de detalii de utilizare. Problemele abstractizării datelor, a modularităţii sau a tratării erorilor sunt deseori ignorate, pentru a transmite cât mai concis esenţa algoritmului.

Sortarea prin inserţie

Începem cu ***sortarea prin inserţie***, care este un algoritm eficient pentru sortarea unui număr mic de obiecte. Sortarea prin inserţie funcţionează în acelaşi fel în care mulţi oameni sortează un pachet de cărţi de joc obişnuite. Se începe cu pachetul aşezat pe masă cu faţa în jos şi cu mâna stângă goală. Apoi, luăm câte o carte de pe masă şi o inserăm în poziţia corectă în mâna stângă. Pentru a găsi poziţia corectă pentru o carte dată, o comparăm cu fiecare dintre cărţile aflate deja în mâna stângă, de la dreapta la stânga, aşa cum este ilustrat în figura 1.1.

Pseudocodul pentru sortarea prin inserţie este prezentat ca o procedură numită SORTEAZĂ-PRIN-INSERŢIE, care are ca parametru un vector $A[1..n]$ conţinând un şir de lungime n care urmează a fi sortat. (Pe parcursul codului, numărul de elemente ale lui A este notat prin *lungime*$[A]$.) Numerele de intrare sunt ***sortate pe loc***, în cadrul aceluiaşi vector A; cel mult un număr constant dintre acestea sunt memorate în zone de memorie suplimentare. Când SORTEAZĂ-PRIN-INSERŢIE se termină, vectorul iniţial A va conţine elementele şirului de ieşire sortat.

SORTEAZĂ-PRIN-INSERŢIE(A)

```
1: pentru j ← 2, lungime[A] execută
2:    cheie ← A[j]
3:    ▷ Inserează A[j] în şirul sortat A[1..j − 1]
4:    i ← j − 1
5:    cât timp i > 0 şi A[i] > cheie execută
6:       A[i + 1] ← A[i]
7:       i ← i − 1
8:    A[i + 1] ← cheie
```

Figura 1.2 ilustrează modul de funcţionare a acestui algoritm pentru $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Indicele j corespunde "cărţii" care urmează a fi inserată în mâna stângă. Elementele $A[1..j-1]$ corespund mulţimii de cărţi din mână, deja sortate, iar elementele $A[j+1..n]$ corespund pachetului de cărţi aflate încă pe masă. Indicele se deplasează de la stânga la dreapta în interiorul vectorului. La fiecare iteraţie, elementul $A[j]$ este ales din vector (linia 2). Apoi, plecând de la poziţia $j-1$, elementele sunt, succesiv, deplasate o poziţie spre dreapta până când este găsită poziţia corectă pentru $A[j]$ (liniile 4–7), moment în care acesta este inserat (linia 8).

Convenţii pentru pseudocod

La scrierea pseudocodului vom folosi următoarele convenţii:

1. Indentarea indică o structură de bloc. De exemplu, corpul ciclului **pentru**, care începe în linia 1, constă din liniile 2–8 iar corpul ciclului **cât timp**, care începe în linia 5, conţine liniile 6–7, dar nu şi linia 8. Stilul nostru de indentare se aplică şi structurilor de tipul **dacă-atunci-altfel**. Folosirea indentării în locul unor indicatori de bloc de tipul **begin** şi **end**, reduce cu mult riscul de confuzie, îmbunătăţind claritatea prezentării.[1]

2. Ciclurile de tipul **cât timp**, **pentru**, **repetă** şi construcţiile condiţionale **dacă**, **atunci** şi **altfel** au aceeaşi interpretare ca şi structurile similare din Pascal.

3. Simbolul ▷ indică faptul că restul liniei este un comentariu.

4. O atribuire multiplă de forma $i \leftarrow j \leftarrow e$ înseamnă atribuirea valorii expresiei e ambelor variabile i şi j; aceasta ar trebui tratată ca un echivalent al atribuirii $j \leftarrow e$, urmată de atribuirea $i \leftarrow j$.

[1] În limbajele de programare reale, indentarea, ca unică metodă pentru indicarea structurilor de bloc, nu este în general recomandabilă, deoarece nivelurile de indentare se determină greoi în cazul unor coduri care se continuă pe mai multe pagini.

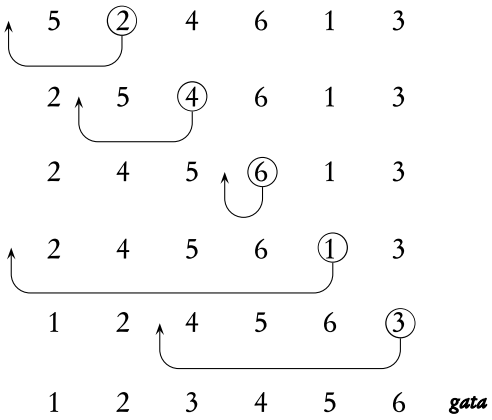


Figura 1.2 Modul de operare a procedurii SORTEAZĂ-PRIN-INSERŢIE asupra vectorului $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Poziţia indicelui j este indicată printr-un cerc.

5. Variabilele (de exemplu i, j, *cheie*) sunt locale pentru o procedură dată. Nu vom utiliza variabile globale fără a preciza acest lucru în mod explicit.

6. Elementele unui vector sunt accesate specificând numele vectorului urmat de indice în paranteze drepte. De exemplu, $A[i]$ indică elementul de rang i al vectorului A. Notaţia ".." este folosită pentru a indica un domeniu de valori în cadrul unui vector. Astfel, $A[1..j]$ indică un subvector al lui A constând din elementele $A[1], A[2], \ldots, A[j]$.

7. Datele compuse sunt, în mod uzual, organizate în ***obiecte*** care conţin ***atribute*** sau ***câmpuri***. Un anumit câmp este accesat folosind numele câmpului urmat de numele obiectului său în paranteze drepte. De exemplu, tratăm un vector ca pe un obiect cu atributul *lungime* indicând numărul de elemente ale acestuia. Pentru a specifica numărul de elemente ale unui vector A, se va scrie *lungime*$[A]$. Deşi vom folosi parantezele drepte atât pentru indexarea elementelor unui vector, cât şi pentru atributele obiectelor, va fi clar din context care este interpretarea corectă.

   O variabilă reprezentând un vector sau un obiect este tratată ca un pointer spre datele care reprezintă vectorul sau obiectul. Pentru toate câmpurile f ale unui obiect x, atribuirea $y \leftarrow x$ are ca efect $f[y] = f[x]$. Mai mult, dacă acum avem $f[x] \leftarrow 3$, atunci nu numai $f[x] = 3$, dar, în acelaşi timp, avem si $f[y] = 3$. Cu alte cuvinte, x şi y indică spre (sau "sunt") acelaşi obiect după atribuirea $y \leftarrow x$. Uneori, un pointer nu se va referi la nici un obiect. În acest caz special pointerul va primi valoarea NIL.

8. Parametrii sunt transmişi unei proceduri ***prin valoare***: procedura apelată primeşte propria sa copie a parametrilor şi, dacă atribuie o valoare unui parametru, schimbarea *nu* este văzută de procedura apelantă. Când obiectele sunt transmise procedurii, este copiat doar pointerul spre datele reprezentând obiectul, nu şi câmpurile acestuia. De exemplu, dacă x este un parametru al unei proceduri apelate, atribuirea $x \leftarrow y$ în cadrul procedurii apelate nu este vizibilă din procedura apelantă. Atribuirea $f[x] \leftarrow 3$ este, totuşi, vizibilă.

Exerciţii

1.1-1 Folosind ca model figura 1.2, ilustraţi modul de operare al procedurii SORTEAZĂ-PRIN-INSERŢIE asupra vectorului $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

1.1-2 Rescrieţi procedura SORTEAZĂ-PRIN-INSERŢIE pentru a sorta în ordine necrescătoare în loc de ordine nedescrescătoare.

1.1-3 Să considerăm ***problema căutării***:

Date de intrare: Un şir de n numere $A = \langle a_1, a_2, \ldots, a_n \rangle$ şi o valoare v.

Date de ieşire: Un indice i astfel încât $v = A[i]$ sau valoarea specială NIL dacă v nu apare în A.

Scrieţi un algoritm pentru o ***căutare liniară***, care parcurge şirul în căutarea lui v.

1.1-4 Să considerăm problema adunării a doi întregi binari pe n biţi memoraţi în doi vectori n-dimensionali A şi B. Suma celor doi întregi trebuie să fie memorată în formă binară într-un vector C având $n+1$ elemente. Găsiţi un enunţ formal al problemei şi scrieţi un algoritm pentru adunarea celor doi întregi.

1.2. Analiza algoritmilor

Analiza unui algoritm a ajuns să însemne prevederea resurselor pe care algoritmul le solicită. Uneori, resurse ca memoria, lăţimea benzii de comunicaţie, porţi logice sunt prima preocupare, dar, de cele mai multe ori, vrem să măsurăm timpul de execuţie necesar algoritmului. În general, analizând mai mulţi algoritmi pentru o anumită problemă, cel mai eficient poate fi identificat uşor. O astfel de analiză poate indica mai mulţi candidaţi viabili, dar câţiva algoritmi inferiori sunt, de obicei, eliminaţi în timpul analizei.

Înainte de a analiza un algoritm, trebuie să avem un model al tehnologiei de implementare care urmează să fie folosită, incluzând un model pentru resursele acesteia şi costurile corespunzătoare. În cea mai mare parte a acestei cărţi, vom presupune că tehnologia de implementare este un model de calcul generic cu un procesor, ***maşină cu acces aleator (random-access machine, RAM)*** şi vom presupune că algoritmii vor fi implementaţi ca programe pentru calculator. În modelul RAM, instrucţiunile sunt executate una după alta, fără operaţii concurente. Totuşi, în partea finală a cărţii, vom avea ocazia să investigăm modele de calcul paralel şi hardware digital.

Analiza, chiar şi a unui singur algoritm, poate fi, uneori, o încercare dificilă. Matematica necesară poate să includă combinatorică, teoria probabilităţilor, dexteritate algebrică şi abilitatea de a identifica cei mai importanţi termeni într-o expresie. Deoarece comportarea unui algoritm poate fi diferită în funcţie de datele de intrare, avem nevoie de un mijloc de exprimare a acestei comportări în formule simple, uşor de înţeles.

Deşi, pentru a analiza un algoritm, selectăm numai un anumit tip de model de maşină de calcul, rămân mai multe posibilităţi de alegere a felului în care decidem să exprimăm această analiză. Un scop imediat este de a găsi un mijloc de exprimare care să fie simplu de scris şi de manevrat, care să arate principalele resurse necesare unui algoritm şi să suprime detaliile inutile.

Analiza sortării prin inserţie

Timpul de execuţie necesar procedurii SORTEAZĂ-PRIN-INSERŢIE depinde de intrare: sortarea a o mie de numere ia mai mult timp decât sortarea a trei. Mai mult decât atât, SORTEAZĂ-PRIN-INSERŢIE poate să consume timpi diferiţi pentru a sorta două şiruri de numere de aceeaşi dimensiune, în funcţie de măsura în care acestea conţin numere aproape sortate. În general, timpul necesar unui algoritm creşte o dată cu dimensiunea datelor de intrare, astfel încât este tradiţional să se descrie timpul de execuţie al unui program în funcţie de dimensiunea datelor de intrare. În acest scop, trebuie să definim cu mai multă precizie termenii de "timp de execuţie" şi "dimensiune a datelor de intrare".

Definiţia ***dimensiunii datelor de intrare*** depinde de problema studiată. Pentru multe probleme, cum ar fi sortarea sau calculul unei transformate Fourier discrete, cea mai naturală măsură este *numărul de obiecte din datele de intrare* – de exemplu, pentru sortare, un vector de dimensiune n. Pentru multe alte probleme, ca spre exemplu înmulţirea a doi întregi, cea mai bună măsură pentru dimensiunea datelor de intrare este *numărul total de biţi* necesari pentru reprezentarea datelor de intrare în notaţie binară. Uneori, este mai potrivit să exprimăm dimensiunea datelor de intrare prin două numere în loc de unul. De exemplu, dacă datele de intrare ale unui algoritm sunt reprezentate de un graf, dimensiunea datelor de intrare poate fi descrisă prin numărul de vârfuri şi muchii ale grafului. Pentru fiecare problemă pe care o vom studia, vom indica măsura utilizată pentru dimensiunea datelor de intrare.

Timpul de execuţie a unui algoritm pentru un anumit set de date de intrare este determinat de numărul de operaţii primitive sau "paşi" executaţi. Este util să definim noţiunea de "pas" astfel încât să fie cât mai independent de calculator. Pentru moment, să adoptăm următorul punct de vedere. Pentru execuţia unei linii din pseudocod este necesară o durată constantă de timp. O anumită linie poate avea nevoie de un timp de execuţie diferit decât o alta, dar vom presupune că fiecare execuţie a liniei i consumă timpul c_i, unde c_i este o constantă. Acest punct de vedere este conform cu modelul RAM şi, în acelaşi timp, reflectă, destul de bine, modul în care pseudocodul poate fi, de fapt, utilizat în cele mai multe cazuri concrete.[2]

În prezentarea care urmează, expresia noastră pentru timpul de execuţie al algoritmului SORTEAZĂ-PRIN-INSERŢIE va evolua de la o formulă relativ complicată, care foloseşte toate costurile de timp c_i, la una mult mai simplă în notaţii, care este mai concisă şi mai uşor de manevrat. Această notaţie mai simplă va face, de asemenea, uşor de determinat dacă un algoritm este mai eficient decât altul.

Începem prin a relua prezentarea procedurii SORTEAZĂ-PRIN-INSERŢIE, adăugând "costul" de timp pentru fiecare instrucţiune şi un număr care reprezintă de câte ori aceasta este efectiv executată. Pentru fiecare $j = 2, 3, \ldots, n$, unde $n = \mathit{lungime}[A]$, vom nota cu t_j numărul de execuţii ale testului **cât timp** din linia 5 pentru valoarea fixată j. Vom presupune că un comentariu nu este o instrucţiune executabilă, prin urmare nu cere timp de calcul.

Timpul de execuţie al algoritmului este suma tuturor timpilor de execuţie corespunzători

[2] Există aici câteva subtilităţi: paşii de calcul pe care îi precizăm sunt, cel mai adesea, variante ale unor proceduri care cer mai mult decât doar un timp constant. De exemplu, în continuare, în această carte am putea spune, într-o linie de pseudocod, "sortează punctele conform coordonatei x", care, aşa cum vom vedea, cere mai mult decât un timp constant. De asemenea, se poate observa că o instrucţiune care apelează o subrutină are nevoie de un timp constant, deşi subrutina, o dată apelată, are nevoie de mai mult timp de execuţie. Din acest motiv, separăm procesul de a ***apela*** o subrutină – trecerea parametrilor către aceasta etc. – de procesul de a ***executa*** subrutina.

SORTEAZĂ-PRIN-INSERŢIE(A)	*cost*	*timp*
1: **pentru** $j \leftarrow 2$, *lungime*$[A]$ **execută**	c_1	n
2: *cheie* $\leftarrow A[j]$	c_2	$n-1$
3: $\triangleright$ Inserează $A[j]$ în şirul sortat $A[1..j-1]$	0	$n-1$
4: $i \leftarrow j-1$	c_4	$n-1$
5: **cât timp** $i > 0$ şi $A[i] >$ *cheie* **execută**	c_5	$\sum_{j=2}^{n} t_j$
6: $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7: $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8: $A[i+1] \leftarrow$ *cheie*	c_8	$n-1$

fiecărei instrucţiuni executate: o instrucţiune care consumă timpul c_i pentru execuţie şi este executată de n ori, va contribui cu $c_i n$ la timpul total de execuţie.[3] Pentru a calcula $T(n)$, timpul de execuţie pentru SORTEAZĂ-PRIN-INSERŢIE, vom aduna produsele mărimilor indicate în coloanele *cost* şi *timp*, obţinând

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) \\ &+ c_8(n-1). \end{aligned}$$

Chiar pentru date de intrare de aceeaşi mărime, timpul de execuţie al unui algoritm dat poate să depindă de *conţinutul* datelor de intrare. De exemplu, pentru SORTEAZĂ-PRIN-INSERŢIE, cazul cel mai favorabil apare când vectorul de intrare este deja sortat. Pentru fiecare $j = 2, 3 \ldots, n$, vom găsi că $A[i] \leq$ *cheie* în linia 5, când i are valoarea iniţială $j-1$. Rezultă $t_j = 1$ pentru $j = 2, 3, \ldots, n$ şi timpul de execuţie în cazul cel mai favorabil este

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

Acest timp de execuţie poate fi exprimat sub forma $an + b$ pentru anumite *constante* a şi b care depind doar de timpii de execuţie c_i, fiind astfel o ***funcţie liniară*** de n.

Dacă vectorul este sortat în ordine inversă – adică, în ordine descrescătoare – obţinem cazul cel mai defavorabil. În această situaţie trebuie să comparăm fiecare element $A[j]$ cu fiecare element din subvectorul $A[1..j-1]$, şi, astfel, $t_j = j$ pentru $j = 2, 3, \ldots, n$. Observând că

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

şi

$$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

[3] Un fenomen de acest tip nu mai are loc atunci când se referă la alte resurse, cum ar fi, de exemplu, memoria. O instrucţiune care se referă la m cuvinte din memorie şi este executată de n ori, nu consumă în general mn cuvinte de memorie.

(vom reveni asupra acestor sume în capitolul 3), găsim că în cazul cel mai defavorabil timpul de execuţie pentru SORTEAZĂ-PRIN-INSERŢIE este

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) \\ &+ c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &- (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

Rezultă că timpul de execuţie în cazul cel mai defavorabil poate fi exprimat sub forma $an^2 + bn + c$, unde constantele a, b şi c depind, din nou, de costurile c_i ale instrucţiunilor, fiind astfel o ***funcţie pătratică*** de n.

De obicei, la fel ca la sortarea prin inserţie, timpul de execuţie al unui algoritm dat este fix pentru anumite date de intrare. Totuşi, în ultimele capitole, vom întâlni câţiva algoritmi "aleatori" a căror comportare poate varia chiar pentru aceleaşi date de intrare.

Analiza celui mai defavorabil caz şi a cazului mediu

În analiza sortării prin inserţie am cercetat ambele situaţii extreme: cazul cel mai favorabil, în care vectorul de intrare era deja sortat, respectiv cel mai defavorabil, în care vectorul de intrare era sortat în ordine inversă. În continuare (pe tot parcursul acestei cărţi), ne vom concentra, de regulă, pe găsirea ***timpului de execuţie în cazul cel mai defavorabil***, cu alte cuvinte, a celui mai mare timp de execuţie posibil relativ la *orice* date de intrare de dimensiune constantă n. Precizăm trei motive pentru această orientare:

- Timpul de execuţie al unui algoritm în cazul cel mai defavorabil este o margine superioară a timpului de execuţie pentru orice date de intrare de dimensiune fixă. Cunoscând acest timp, avem o garanţie că algoritmul nu va avea, niciodată, un timp de execuţie mai mare. Nu va fi nevoie să facem presupuneri sau investigaţii suplimentare asupra timpului de execuţie şi să sperăm că acesta nu va fi, niciodată, mult mai mare.

- Pentru anumiţi algoritmi, cazul cel mai defavorabil apare destul de frecvent. De exemplu, în căutarea unei anumite informaţii într-o bază de date, cazul cel mai defavorabil al algoritmului de căutare va apare deseori când informaţia căutată nu este de fapt prezentă în baza de date. În anumite aplicaţii, căutarea unor informaţii absente poate fi frecventă.

- "Cazul mediu" este, adesea, aproape la fel de defavorabil ca şi cazul cel mai defavorabil. Să presupunem că alegem la întâmplare n numere şi aplicăm sortarea prin inserţie. Cât timp va fi necesar pentru a determina locul în care putem insera $A[j]$ în subvectorul $A[1..j-1]$? În medie, jumătate din elementele subvectorului $A[1..j-1]$ sunt mai mici decât $A[j]$, şi cealaltă jumătate sunt mai mari. Prin urmare, în medie, trebuie verificate jumătate din elementele subvectorului $A[1..j-1]$, deci $t_j = j/2$. Dacă ţinem seama de această observaţie, timpul de execuţie mediu va apărea tot ca o funcţie pătratică de n, la fel ca în cazul cel mai defavorabil.

În anumite cazuri particulare, vom fi interesaţi de ***timpul mediu de execuţie*** al unui algoritm. O problemă care apare în analiza cazului mediu este aceea că s-ar putea să nu fie prea clar din ce sunt constituite datele de intrare "medii" pentru o anumită problemă. Adesea, vom presupune că toate datele de intrare având o dimensiune dată sunt la fel de probabile. În practică, această presupunere poate fi falsă, dar un algoritm aleator poate, uneori, să o forţeze.

Ordinul de creştere

Pentru a uşura analiza procedurii SORTEAZĂ-PRIN-INSERŢIE, am utilizat mai multe presupuneri simplificatoare. În primul rând, am ignorat costul real al fiecărei instrucţiuni, folosind constantele c_i pentru a reprezenta aceste costuri. Apoi, am observat că, prin aceste constante, obţinem mai multe detalii decât avem nevoie în mod real: timpul de execuţie în cazul cel mai defavorabil este de forma an^2+bn+c pentru anumite constante a, b şi c care depind de costurile c_i ale instrucţiunilor. Astfel, am ignorat nu numai costurile reale ale instrucţiunilor, dar şi costurile abstracte c_i.

Vom face acum încă o abstractizare simplificatoare. Ceea ce ne interesează de fapt, este ***rata de creştere*** sau ***ordinul de creştere*** a timpului de execuţie. Considerăm, prin urmare, doar termenul dominant al formulei (adică an^2) deoarece ceilalţi termeni sunt relativ nesemnificativi pentru valori mari ale lui n. Ignorăm, de asemenea, şi factorul constant c, deoarece, pentru numere foarte mari, factorii constanţi sunt mai puţin semnificativi decât rata de creştere în determinarea eficienţei computaţionale a unor algoritmi. Astfel, vom spune, de exemplu, că sortarea prin inserţie are un timp de execuţie în cazul cel mai defavorabil de $\Theta(n^2)$ (pronunţat "teta de n pătrat"). Vom folosi notaţia de tip Θ în acest capitol cu caracter informal; va fi definită cu precizie în capitolul 2.

În mod uzual, vom considera un algoritm ca fiind mai eficient decât altul dacă timpul său de execuţie în cazul cel mai defavorabil are un ordin de creştere mai mic. Această evaluare ar putea fi incorectă pentru date de intrare de dimensiune mică, dar în cazul unor date de intrare de dimensiuni foarte mari, un algoritm de tipul $\Theta(n^2)$, de exemplu, va fi executat în cazul cel mai defavorabil mult mai repede decât unul de tipul $\Theta(n^3)$.

Exerciţii

1.2-1 Să considerăm sortarea a n numere memorate într-un vector A, pentru început găsind cel mai mic element al lui A şi punându-l ca prim element într-un alt vector B. Apoi, găsim al doilea element mai mic din A şi îl punem pe poziţia a doua a lui B. Continuaţi în acelaşi mod pentru toate elementele lui A. Scrieţi un pseudocod pentru acest algoritm, care este cunoscut sub numele de ***sortarea prin selecţie***. Găsiţi timpii de execuţie în cazul cel mai favorabil, respectiv cel mai defavorabil, pentru sortarea prin selecţie, utilizând notaţia Θ.

1.2-2 Să considerăm, din nou, căutarea liniară (vezi exerciţiul 1.1-3). Cât de multe elemente ale şirului de intrare trebuie verificate, în medie, presupunând că elementul căutat se află printre termenii şirului dat? Ce puteţi spune despre cazul cel mai defavorabil? Care sunt timpul mediu de execuţie şi timpul de execuţie în cazul cel mai defavorabil pentru căutarea liniară, exprimaţi în notaţia Θ? Justificaţi răspunsurile.

1.2-3 Să considerăm problema evaluării unui polinom într-un punct. Fiind daţi n coeficienţi $a_0, a_1, \ldots, a_{n-1}$ şi un număr real x, dorim să calculăm $\sum_{i=0}^{n-1} a_i x^i$. Descrieţi un algoritm simplu

în timp $\Theta(n^2)$ pentru această operaţie. Descrieţi şi un algoritm în timp $\Theta(n)$ care foloseşte următoarea metodă de rescriere a unui polinom, numită schema lui Horner:

$$\sum_{i=0}^{n-1} a_i x^i = (\cdots(a_{n-1}x + a_{n-2})x + \cdots + a_1)x + a_0$$

1.2-4 Scrieţi funcţia $n^3/1000 - 100n^2 - 100n + 3$ cu ajutorul notaţiei Θ.

1.2-5 Cum poate fi modificat aproape orice algoritm pentru a avea un timp de execuţie bun în cel mai favorabil caz?

1.3. Proiectarea algoritmilor

Există multe moduri de proiectare a algoritmilor. Sortarea prin inserţie foloseşte o metodă care s-ar putea numi ***incrementală***: având deja sortat subvectorul $A[1..j-1]$, inserăm elementul $A[j]$ în locul potrivit producând subvectorul $A[1..j]$.

În această secţiune vom examina o abordare diferită, numită *divide şi stăpâneşte*. Vom utiliza această abordare pentru a construi un algoritm de sortare al cărui timp de execuţie în cazul cel mai defavorabil va fi mult mai mic decât al celui corespunzător sortării prin inserţie. Unul din avantajele algoritmilor de tipul *divide şi stăpâneşte* este acela că timpul lor de execuţie este adesea uşor de determinat folosind tehnici care vor fi introduse în capitolul 4.

1.3.1. Abordarea divide şi stăpâneşte

Mulţi algoritmi utili au o structură ***recursivă***: pentru a rezolva o problemă dată, aceştia sunt apelaţi de către ei înşişi o dată sau de mai multe ori pentru a rezolva subprobleme apropiate. Aceşti algoritmi folosesc de obicei o abordare de tipul ***divide şi stăpâneşte***: ei rup problema de rezolvat în mai multe probleme similare problemei iniţiale, dar de dimensiune mai mică, le rezolvă în mod recursiv şi apoi le combină pentru a crea o soluţie a problemei iniţiale.

Paradigma *divide şi stăpâneşte* implică trei paşi la fiecare nivel de recursivitate:

Divide problema într-un număr de subprobleme.

Stăpâneşte subproblemele prin rezolvarea acestora în mod recursiv. Dacă dimensiunile acestora sunt suficient de mici, rezolvă subproblemele în modul uzual, nerecursiv.

Combină soluţiile tuturor subproblemelor în soluţia finală pentru problema iniţială.

Algoritmul de ***sortare prin interclasare*** urmează îndeaproape paradigma *divide şi stăpâneşte*. Intuitiv, acesta operează astfel.

Divide: Împarte şirul de n elemente care urmează a fi sortat în două subşiruri de câte $n/2$ elemente.

Stăpâneşte: Sortează recursiv cele două subşiruri utilizând sortarea prin interclasare.

Combină: Interclasează cele două subşiruri sortate pentru a produce rezultatul final.

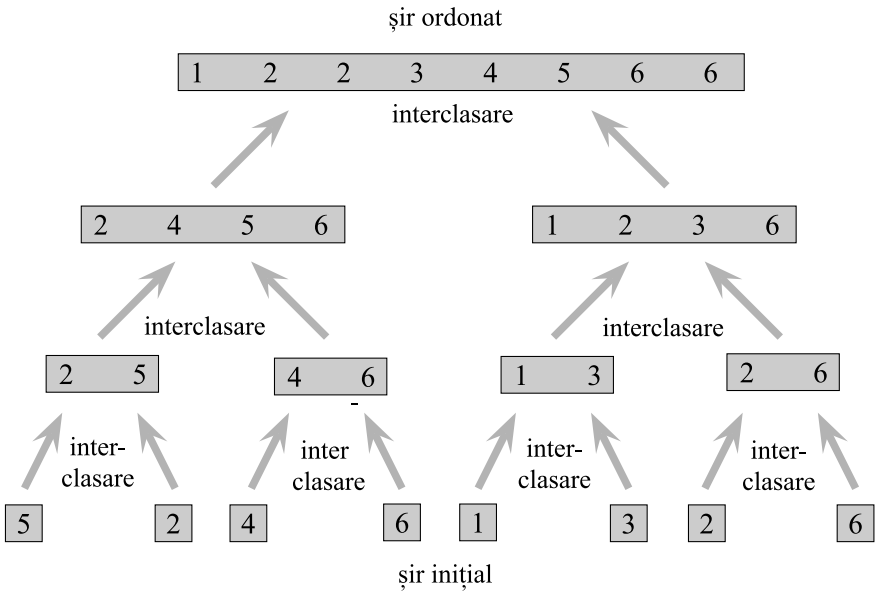


Figura 1.3 Modul de operare al sortării prin interclasare asupra vectorului $A = \langle 5, 2, 4, 6, 1, 3, 2, 6 \rangle$. Lungimile şirurilor sortate, în curs de interclasare, cresc pe măsură ce algoritmul avansează de jos în sus.

Să observăm că recursivitatea se opreşte când şirul de sortat are lungimea 1, caz în care nu mai avem nimic de făcut, deoarece orice şir de lungime 1 este deja sortat.

Operaţia principală a algoritmului de sortare prin interclasare este interclasarea a două şiruri sortate, în pasul denumit mai sus "Combină". Pentru aceasta vom utiliza o procedură auxiliară, INTERCLASEAZĂ(A, p, q, r), unde A este un vector şi p, q şi r sunt indici ai vectorului, astfel încât $p \leq q < r$. Procedura presupune că subvectorii $A[p..q]$ şi $A[q+1..r]$ sunt sortaţi. Ea îi ***interclasează*** pentru a forma un subvector sortat care înlocuieşte subvectorul curent $A[p..r]$.

Deşi vom lăsa pseudocodul pentru această procedură ca exerciţiu (vezi exerciţiul 1.3-2), este uşor de imaginat o procedură de tip INTERCLASEAZĂ al cărei timp de execuţie este de ordinul $\Theta(n)$, în care $n = r - p + 1$ este numărul elementelor interclasate. Revenind la exemplul nostru cu cărţile de joc, să presupunem că avem două pachete de cărţi de joc aşezate pe masă cu faţa în sus. Fiecare din cele două pachete este sortat, cartea cu valoarea cea mai mică fiind deasupra. Dorim să amestecăm cele două pachete într-un singur pachet sortat, care să rămână aşezat pe masă cu faţa în jos. Pasul principal este acela de a selecta cartea cu valoarea cea mai mică dintre cele două aflate deasupra pachetelor (fapt care va face ca o nouă carte să fie deasupra pachetului respectiv) şi de a o pune cu faţa în jos pe locul în care se va forma pachetul sortat final. Repetăm acest procedeu până când unul din pachete este epuizat. În această fază, este suficient să luăm pachetul rămas şi să-l punem peste pachetul deja sortat întorcând toate cărţile cu faţa în jos. Din punctul de vedere al timpului de execuţie, fiecare pas de bază durează un timp constant, deoarece comparăm de fiecare dată doar două cărţi. Deoarece avem de făcut cel mult n astfel de operaţii elementare, timpul de execuţie pentru procedura INTERCLASEAZĂ este $\Theta(n)$.

Acum, putem utiliza procedura INTERCLASEAZĂ ca subrutină pentru algoritmul de sortare

SORTEAZĂ-PRIN-INTERCLASARE(A, p, r)
1: **dacă** $p < r$ **atunci**
2: $q \leftarrow \lfloor (p + r)/2 \rfloor$
3: SORTEAZĂ-PRIN-INTERCLASARE(A, p, q)
4: SORTEAZĂ-PRIN-INTERCLASARE$(A, q + 1, r)$
5: INTERCLASEAZĂ(A, p, q, r)

prin interclasare. Procedura SORTEAZĂ-PRIN-INTERCLASARE(A, p, r) sortează elementele din subvectorul $A[p..r]$. Dacă $p \geq r$, subvectorul are cel mult un element şi este, prin urmare, deja sortat. Altfel, pasul de divizare este prezent aici prin simplul calcul al unui indice q care împarte $A[p..r]$ în doi subvectori, $A[p..q]$ conţinând $\lceil n/2 \rceil$ elemente şi $A[q + 1..r]$ conţinând $\lfloor n/2 \rfloor$ elemente.[4] Pentru a sorta întregul şir $A = \langle A[1], A[2], \ldots A[n] \rangle$, vom apela procedura SORTEAZĂ-PRIN-INTERCLASARE sub forma SORTEAZĂ-PRIN-INTERCLASARE$(A, 1, \mathit{lungime}[A])$ unde, din nou, $\mathit{lungime}[A] = n$. Dacă analizăm modul de operare al procedurii, de jos în sus, când n este o putere a lui 2, algoritmul constă din interclasarea perechilor de şiruri de lungime 1, pentru a forma şiruri sortate de lungime 2, interclasarea acestora în şiruri sortate de lungime 4, şi aşa mai departe, până când două şiruri sortate de lungime $n/2$ sunt interclasate pentru a forma şirul sortat final de dimensiune n. Figura 1.3 ilustrează acest proces.

1.3.2. Analiza algoritmilor de tipul divide şi stăpâneşte

Când un algoritm conţine un apel recursiv la el însuşi, timpul său de execuţie poate fi, adesea, descris printr-o ***relaţie de recurenţă*** sau, mai simplu, ***recurenţă***, care descrie întregul timp de execuţie al unei probleme de dimensiune n cu ajutorul timpilor de execuţie pentru date de intrare de dimensiuni mai mici. Putem, apoi, folosi instrumente matematice pentru a rezolva problema de recurenţă şi pentru a obţine margini ale performanţei algoritmului.

O recurenţă pentru timpul de execuţie al unui algoritm de tipul divide-et-impera se bazează pe cele trei etape definite în descrierea metodei. La fel ca până acum, vom nota cu $T(n)$ timpul de execuţie al unei probleme de mărime n. Dacă dimensiunea problemei este suficient de mică, de exemplu $n \leq c$ pentru o anumită constantă c, soluţia directă ia un timp constant de execuţie, pe care îl vom nota cu $\Theta(1)$. Să presupunem că divizăm problema în a subprobleme, fiecare dintre acestea având dimensiunea de $1/b$ din dimensiunea problemei originale. Dacă $D(n)$ este timpul necesar pentru a divide problema în subprobleme, iar $C(n)$ este timpul necesar pentru a combina soluţiile subproblemelor în soluţia problemei originale, obţinem recurenţa

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{în caz contrar.} \end{cases}$$

În capitolul 4 vom vedea cum se pot rezolva recurenţe uzuale pentru probleme de această formă.

Analiza sortării prin interclasare

Deşi algoritmul SORTEAZĂ-PRIN-INTERCLASARE funcţionează corect când numărul elementelor nu este par, analiza bazată pe recurenţă se simplifică dacă presupunem că dimensiunea

[4] Notaţia $\lceil x \rceil$ desemnează cel mai mic număr întreg mai mare sau egal decât x şi $\lfloor x \rfloor$ desemnează cel mai mare număr întreg mai mic sau egal decât x. Aceste notaţii sunt definite în capitolul 2.

problemei originale este o putere a lui 2. Fiecare pas de împărţire generează deci două subşiruri având dimensiunea exact $n/2$. În capitolul 4 vom vedea că această presupunere nu afectează ordinul de creştere a soluţiei recurenţei.

Pentru a determina recurenţa pentru $T(n)$, timpul de execuţie al sortării prin interclasare a n numere în cazul cel mai defavorabil, vom raţiona în felul următor. Sortarea prin interclasare a unui singur element are nevoie de un timp constant. Când avem $n > 1$ elemente, vom descompune timpul de execuţie după cum urmează:

Divide: La acest pas, se calculează doar mijlocul subvectorului, calcul care are nevoie de un timp constant de execuţie. Astfel, $D(n) = \Theta(1)$.

Stăpâneşte: Rezolvăm recursiv două subprobleme, fiecare de dimensiune $n/2$, care contribuie cu $2T(n/2)$ la timpul de execuţie.

Combină: Am observat deja că procedura INTERCLASEAZĂ pentru un subvector cu n elemente consumă $\Theta(n)$ timp de execuţie, deci $C(n) = \Theta(n)$.

Când adunăm funcţiile $D(n)$ şi $C(n)$ pentru analiza sortării prin interclasare, adunăm o funcţie cu timpul de execuţie $\Theta(n)$ cu o funcţie cu timpul de execuţie $\Theta(1)$. Această sumă este funcţie liniară în raport cu n, adică are timpul de execuţie $\Theta(n)$. Adăugând aceasta la termenul $2T(n/2)$ de la pasul "Stăpâneşte", obţinem timpul de execuţie $T(n)$ în cazul cel mai defavorabil pentru sortarea prin interclasare:

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n = 1, \\ 2T(n/2) + \Theta(n) & \text{dacă } n > 1. \end{cases}$$

În capitolul 4, vom arăta că $T(n)$ este $\Theta(n \lg n)$, unde $\lg n$ reprezintă $\log_2 n$. Pentru numere suficient de mari, sortarea prin interclasare, având timpul de execuţie $\Theta(n \lg n)$, este mai performantă decât sortarea prin inserţie, al cărei timp de execuţie în cazul cel mai defavorabil este $\Theta(n^2)$.

Exerciţii

1.3-1 Utilizând ca model figura 1.3, ilustraţi modul de operare al sortării prin interclasare pentru vectorul $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

1.3-2 Scrieţi pseudocodul pentru INTERCLASEAZĂ(A, p, q, r).

1.3-3 Utilizaţi inducţia matematică pentru a arăta că, dacă n este o putere a lui 2, soluţia recurenţei

$$T(n) = \begin{cases} 2 & \text{dacă } n = 2, \\ 2T(n/2) + n & \text{dacă } n = 2^k, k > 1. \end{cases}$$

este $T(n) = n \lg n$.

1.3-4 Sortarea prin inserţie poate fi descrisă ca procedură recursivă în felul următor: pentru a sorta $A[1..n]$, sortăm $A[1..n-1]$ şi apoi inserăm $A[n]$ în vectorul sortat $A[1..n-1]$. Scrieţi o recurenţă pentru timpul de execuţie a acestei versiuni a sortării prin inserţie.

1.3-5 Reluând problema căutării (vezi exerciţiul 1.1-3), observaţi că, dacă şirul A este sortat, se poate începe prin a compara v cu valoarea aflată la mijlocul vectorului şi se poate elimina din discuţie una din jumătăţi. ***Căutarea binară*** este un algoritm care repetă acest procedeu, înjumătăţind de fiecare dată partea şirului în care se face căutarea. Scrieţi un pseudocod fie iterativ, fie recursiv, pentru căutarea binară. Argumentaţi că timpul de execuţie în cazul cel mai defavorabil este $\Theta(\lg n)$.

1.3-6 Observaţi că blocul **cât timp** (liniile 5–7) din procedura SORTEAZĂ-PRIN-INSERŢIE (secţiunea 1.1) foloseşte o căutare liniară pentru a parcurge înapoi subvectorul $A[1..j-1]$. Putem utiliza în locul acesteia o căutare binară (vezi exerciţiul 1.3-5), pentru a îmbunătăţi timpul de execuţie total pentru sortarea prin inserţie în cazul cel mai defavorabil, la valoarea $\Theta(n \lg n)$?

1.3-7 $\star$ Descrieţi un algoritm al cărui timp de execuţie să fie $\Theta(n \lg n)$ şi care, pornind de la o mulţime dată S de n numere reale şi un alt număr real x, să decidă dacă printre elementele lui S există două elemente având suma x.

1.4. Rezumat

Un algoritm bun este ca un cuţit ascuţit – face exact ceea ce se aşteaptă să facă, cu un minimum de efort. A folosi un algoritm nepotrivit pentru a rezolva o problemă este ca şi cum s-ar încerca să se taie o friptură cu o şurubelniţă: în final, s-ar obţine, eventual, un produs digerabil, dar cu un efort considerabil mai mare decât cel necesar, iar rezultatul ar fi, probabil, mai puţin plăcut din punct de vedere estetic.

Algoritmii care sunt proiectaţi să rezolve o aceeaşi problemă pot să difere foarte mult în eficienţă. Aceste diferenţe pot fi cu mult mai semnificative decât diferenţa dintre un calculator personal şi un supercalculator. De exemplu, să ne imaginăm un supercalculator care rulează sortarea prin inserţie şi un calculator personal care rulează sortarea prin interclasare. Fiecare trebuie să sorteze un vector de un milion de numere. Să presupunem că supercalculatorul poate executa 100 milioane de operaţii pe secundă, în timp ce calculatorul personal execută doar un milion de operaţii pe secundă. Pentru a face ca diferenţa să fie şi mai dramatică, vom presupune că supercalculatorul utilizează pentru sortarea prin inserţie un program executabil realizat în cod maşină, optimizat de către cel mai bun programator, rezultatul fiind că sunt necesare $2n^2$ instrucţiuni la supercalculator pentru a realiza sortarea a n numere. Sortarea prin interclasare, pe de altă parte, se rulează cu un program realizat pentru calculatorul personal, de către un programator de nivel mediu, folosind un limbaj de nivel înalt cu un compilator ineficient, codului rezultat fiindu-i necesare $50n \lg n$ instrucţiuni. Pentru a sorta un milion de numere, supercalculatorul are nevoie de

$$\frac{2 \cdot (10^6)^2 \text{ instrucţiuni}}{10^8 \text{ instrucţiuni/secundă}} = 20.000 \text{ secunde } \approx 5,56 \text{ ore},$$

în timp ce pentru calculatorul personal sunt necesare

$$\frac{50 \cdot 10^6 \lg 10^6 \text{ instrucţiuni}}{10^6 \text{ instrucţiuni/secundă}} \approx 1.000 \text{ secunde } \approx 16,67 \text{ minute}.$$

Prin utilizarea unui algoritm al cărui timp de execuţie are un ordin mai mic de creştere, chiar şi un calculator personal obţine rezultatul cerut de 20 de ori mai repede decât supercalculatorul!

Acest exemplu arată că algoritmii, la fel ca şi echipamentele hardware sunt o ***tehnologie***. Performanţa totală a unui sistem de calcul depinde tot atât de mult de alegerea algoritmilor eficienţi, cât depinde de alegerea unui hardware rapid. Aşa cum are loc o dezvoltare tehnologică rapidă în ceea ce priveşte alte tehnologii legate de calculatoare, tot astfel se dezvoltă şi algoritmii.

Exerciţii

1.4-1 Să presupunem că trebuie să comparăm implementările sortării prin inserţie şi sortării prin interclasare pe acelaşi calculator. Pentru date de intrare de dimensiune n, sortarea prin inserţie se execută în $8n^2$ paşi, în timp ce sortarea prin interclasare se execută în $64n \lg n$ paşi. Pentru ce valori ale lui n sortarea prin inserţie este mai rapidă decât sortarea prin interclasare? Cum s-ar putea rescrie pseudocodul pentru sortarea prin interclasare cu scopul de a o face chiar mai rapidă pentru date de intrare de dimensiuni mici?

1.4-2 Care este cea mai mică valoare a lui n pentru care un algoritm cu timpul de execuţie de $100n^2$ este mai rapid decât un algoritm cu timpul timpul de execuţie de 2^n?

1.4-3 Să considerăm problema de a determina dacă un şir arbitrar de n numere $\langle x_1, x_2, \ldots, x_n \rangle$ conţine cel puţin doi termeni egali. Arătaţi că acest fapt poate fi realizat în $\Theta(n \lg n)$, în care $\lg n$ reprezintă $\log_2 n$.

Probleme

1-1 Comparaţia timpilor de execuţie

Pentru fiecare funcţie $f(n)$ şi timp t din următorul tabel, determinaţi cea mai mare valoare a lui n pentru o problemă care poate fi rezolvată în timpul t, presupunând că algoritmul de rezolvare a problemei are nevoie de $f(n)$ microsecunde.

	1 secundă	1 minut	1 oră	1 zi	1 lună	1 an	1 secol
$\lg n$							
$\sqrt{n}$							
n							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

1-2 Sortarea prin inserţie pe vectori mici în sortarea prin interclasare

Deşi timpul de execuţie al algoritmului de sortare prin interclasare în cazul cel mai defavorabil este $\Theta(n \lg n)$, iar sortarea prin inserţie are un timp de execuţie pentru situaţii similare de $\Theta(n^2)$, factorii constanţi din sortarea prin inserţie pot face ca aceasta să fie, totuşi, mai rapidă pentru valori mici ale lui n. Astfel, are sens să se utilizeze sortarea prin inserţie în cadrul sortării prin

interclasare, dacă subproblemele devin suficient de mici. Să considerăm o modificare a sortării prin interclasare în care n/k subvectori de lungine k sunt sortaţi utilizând sortarea prin inserţie şi apoi interclasaţi folosind procedura standard de sortare prin interclasare, unde k este o valoare care urmează a fi determinată.

a. Arătaţi că cei n/k subvectori, fiecare de lungime k, pot fi sortaţi prin inserţie cu timp de execuţie $\Theta(nk)$ în cazul cel mai defavorabil.

b. Arătaţi că aceşti subvectori pot fi sortaţi prin interclasare cu timp de execuţie $\Theta(n \lg(n/k))$ în cazul cel mai defavorabil.

c. Fiind dat faptul că algoritmul modificat are timp de execuţie $\Theta(nk + n \lg(n/k))$ în cazul cel mai defavorabil, care este cea mai mare valoare asimptotică a lui k (cu notaţia Θ) în funcţie de n, pentru care algoritmul modificat are acelaşi timp de execuţie ca şi sortarea standard prin interclasare?

d. Cum ar trebui ales k în mod practic?

1-3 Inversiuni
Fie un vector $A[1..n]$ de n numere distincte. Dacă $i < j$ şi $A[i] > A[j]$, atunci perechea (i, j) se numeşte ***inversiune*** a lui A.

a. Enumeraţi cele cinci inversiuni ale vectorului $\langle 2, 3, 8, 6, 1 \rangle$.

b. Ce vector având elemente din mulţimea $\{1, 2, \ldots, n\}$ are cele mai multe inversiuni? Cât de multe?

c. Care este relaţia dintre timpul de execuţie al sortării prin inserţie şi numărul de inversiuni din vectorul de intrare? Justificaţi răspunsul.

d. Descrieţi un algoritm care determină numărul de inversiuni din orice permutare de n elemente având timp de execuţie $\Theta(n \lg n)$ în cazul cel mai defavorabil. (*Indicaţie:* Modificaţi sortarea prin interclasare.)

Note bibliografice

Există multe texte excelente în legătură cu subiectul general al algoritmilor, inclusiv cele scrise de Aho, Hopcroft şi Ullman [4, 5], Baase [14], Brassard şi Bratley [33], Horowitz şi Sahni [105], Knuth [121, 122, 123], Manber [142], Mehlhorn [144, 145, 146], Purdom şi Brown [164], Reingold, Nievergelt şi Deo [167], Sedgewick [175] şi Wilf [201]. Unele aspecte mai practice ale proiectării algoritmilor sunt discutate de Bentley [24, 25] şi Gonnet [90].

În 1968, Knuth a publicat primul din cele trei volume cu titlul general *Arta Programării Calculatoarelor* [121, 122, 123]. Primul volum a abordat studiul modern al algoritmilor cu focalizare pe analiza timpului de execuţie, iar această serie a rămas o referinţă semnificativă pentru multe din problematicile prezentate aici. Conform lui Knuth, cuvântul "algoritm" este derivat din numele "al-Khowârizmî", un matematician persan din secolul IX.

Aho, Hopcroft şi Ullman [4] au susţinut analiza asimptotică a algoritmilor ca mijloc de comparare a performanţei relative. Ei au popularizat utilizarea relaţiilor de recurenţă pentru a descrie timpii de execuţie ai algoritmilor recursivi.

Knuth [123] prezintă o tratare enciclopedică a multor algoritmi de sortare. Comparaţia algoritmilor de sortare (capitolul 9) include analize exacte ale numărării paşilor, precum aceea care am efectuat-o aici pentru sortarea prin inserţie. Discuţia lui Knuth despre sortarea prin inserţie conţine câteva variaţii ale algoritmului. Cea mai importantă este sortarea lui Shell, introdusă de D. L. Shell, care utilizează sortarea prin inserţie pe subşiruri periodice ale datelor de intrare pentru a produce un algoritm de sortare mai rapid.

Sortarea prin interclasare este, de asemenea, descrisă de Knuth. Acesta menţionează că J. von Neumann a inventat în anul 1938 un mecanism capabil să interclaseze două pachete de cartele perforate într-un singur pas. J. von Neumann, unul dintre pionierii informaticii, se pare că a scris în 1945 un program pentru sortarea prin interclasare pe calculatorul EDVAC.

I Fundamente matematice

Introducere

Analiza algoritmilor necesită deseori utilizarea uneltelor matematice. Aceste unelte uneori sunt foarte simple, cum ar fi algebra de liceu, dar altele, cum ar fi rezolvarea recurenţelor s-ar putea să reprezinte o noutate. Această parte a cărţii este un compendiu al metodelor şi uneltelor pe care în această carte le vom folosi pentru analiza algoritmilor.

Vă sugerăm să nu încercaţi să asimilaţi toată matematica din acest capitol dintr-o dată. Răsfoiţi capitolele din această parte pentru a vedea ce conţin. Apoi puteţi trece direct la capitolele care se ocupă de algoritmi. Pe măsură ce citiţi aceste capitole, reveniţi la această parte oricând aveţi nevoie de o mai bună înţelegere a uneltelor folosite în analizele de factură matematică. Oricum, la un moment dat, veţi dori să studiaţi fiecare din aceste capitole în întregime, astfel încât să vă formaţi fundamente matematice solide.

Capitolul 2 defineşte riguros mai multe notaţii asimptotice, de exemplu notaţia Θ folosită în capitolul 1. În restul capitolului 2 sunt prezentate alte notaţii matematice. Scopul este mai mult de a asigura faptul că modul în care folosiţi notaţiile matematice este acelaşi cu cel folosit în carte, şi nu de a vă învăţa noi noţiuni matematice.

Capitolul 3 oferă metode pentru evaluarea şi mărginirea sumelor, operaţii ce apar foarte des în analiza algoritmilor.

În capitolul 4 sunt prezentate metode de rezolvare a recurenţelor, pe care le-am folosit, de exemplu, pentru a analiza sortarea prin interclasare în capitolul 1 şi cu care ne vom întâlni de multe ori. O tehnică foarte eficientă, care poate fi folosită pentru a rezolva recurenţe ce apar în algoritmi "divide şi stăpâneşte", este "metoda master". O mare parte din capitolul 4 este alocată demonstrării corectitudinii metodei master, deşi cititorul poate sări peste această demonstraţie fără o pierdere prea mare.

Capitolul 5 conţine definiţii şi notaţii de bază pentru mulţimi, relaţii, funcţii, grafuri şi arbori. Acest capitol prezintă de asemenea şi unele proprietăţi de bază ale acestor concepte matematice. Acest material este esenţial pentru înţelegerea cărţii, dar poate fi ignorat fără probleme dacă aţi studiat anterior un curs de matematică discretă.

Capitolul 6 începe cu principii elementare de numărare: permutări, combinări şi alte noţiuni asemănătoare. Restul capitolului conţine definiţii şi proprietăţi elementare de probabilitate. Cei mai mulţi algoritmi din această carte nu necesită estimări probabilistice pentru analiza lor, deci puteţi omite cu uşurinţă ultimele secţiuni la o primă lectură, chiar fără a le răsfoi. Ulterior, când veţi întâlni o analiză probabilistică pe care doriţi să o înţelegeţi mai bine, veţi constata că acest capitol este bine organizat în acest sens.

2 Creşterea funcţiilor

Ordinul de creştere a timpului de execuţie al unui algoritm, definit în capitolul 1 dă o caracterizare simplă pentru eficienţa algoritmilor şi ne dă în acelaşi timp posibilitatea de a compara performanţele relative ale unor algoritmi alternativi. De îndată ce dimensiunea n a datelor de intrare devine suficient de mare, algoritmul de sortare prin interclasare, cu timpul de execuţie $\Theta(n \lg n)$ în cazul cel mai defavorabil, este mai performant decât sortarea prin inserţie, al cărei timp de execuţie în cazul cel mai defavorabil este $\Theta(n^2)$. Deşi uneori se poate determina exact timpul de execuţie al unui algoritm, aşa cum am făcut pentru sortarea prin inserţie în capitolul 1, în general, o precizie suplimentară nu merită efortul făcut. Pentru date de intrare suficient de mari, constantele multiplicative şi termenii de ordin inferior din expresia exactă a unui timp de execuţie sunt dominaţi de efectele dimensiunii datelor de intrare în sine.

Când examinăm date de intrare de dimensiuni suficient de mari, astfel încât numai ordinul de creştere a timpului de execuţie să fie relevant, studiem eficienţa ***asimptotică*** a algoritmilor. Cu alte cuvinte, suntem interesaţi de modul de creştere a timpului de execuţie a unui algoritm o dată cu mărirea dimensiunii datelor de intrare, *în cazul limită* când dimensiunea datelor de intrare creşte nemărginit. În mod uzual, un algoritm care este asimptotic mai eficient va fi cea mai bună alegere, cu excepţia cazului în care datele de intrare sunt de dimensiune foarte mică.

Acest capitol oferă mai multe metode standard pentru simplificarea analizei asimptotice a algoritmilor. Secţiunea următoare începe prin a defini mai multe tipuri de "notaţie asimptotică", dintre care am întâlnit deja Θ-notaţia. Vor fi prezentate mai multe convenţii de notaţie care vor fi utilizate pe parcursul cărţii şi în final vom trece în revistă comportamentul funcţiilor care apar frecvent în analiza algoritmilor.

2.1. Notaţia asimptotică

Notaţiile pe care le utilizăm pentru a descrie timpul asimptotic de execuţie al unui algoritm sunt definite cu ajutorul unor funcţii al căror domeniu de definiţie este de fapt mulţimea numerelor naturale $\mathbb{N} = \{0, 1, 2, \ldots\}$. Astfel de notaţii sunt convenabile pentru a descrie funcţia timp de execuţie în cazul cel mai defavorabil $T(n)$ care în mod uzual este definită doar pentru dimensiuni întregi ale datelor de intrare. Totuşi, este necesar uneori să folosim *în mod abuziv* notaţia asimptotică într-o varietate de moduri. De exemplu, notaţia se extinde cu uşurinţă la domeniul numerelor reale sau, alternativ, se poate restrânge la o submulţime a mulţimii numerelor naturale. Este important totuşi să înţelegem sensul exact al notaţiei pentru ca atunci când este utilizată în mod abuziv să nu fie utilizată în mod *incorect*. În această secţiune se definesc principalele notaţii asimptotice şi de asemenea, se introduc câteva abuzuri de notaţie mai des întâlnite.

Θ-notaţia

În capitolul 1 am arătat că timpul de execuţie pentru sortarea prin inserţie în cazul cel mai defavorabil este $T(n) = \Theta(n^2)$. Să definim acum exact această notaţie. Pentru o funcţie dată

$g(n)$, notăm cu $\Theta(g(n))$, *mulţimea de funcţii*

$$\Theta(g(n)) = \{\, f(n) \;\; : \;\; \text{există constantele pozitive } c_1, c_2 \text{ şi } n_0 \text{ astfel încât} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \;\; \text{pentru orice } n \ge n_0\}.$$

O funcţie $f(n)$ aparţine mulţimii $\Theta(g(n))$ dacă există constantele pozitive c_1 şi c_2 astfel încât aceasta să fie plasată între $c_1 g(n)$ şi $c_2 g(n)$ pentru n suficient de mare. Deşi $\Theta(g(n))$ este o mulţime, vom scrie "$f(n) = \Theta(g(n))$" pentru a indica faptul că $f(n)$ este un membru al acesteia. Acest abuz al utilizării semnului de egalitate pentru a desemna apartenenţa la o mulţime poate să pară confuz la început, dar vom vedea mai târziu că are avantajele sale.

Figura 2.1 (a) dă o imagine intuitivă a funcţiilor $f(n)$ şi $g(n)$, când $f(n) = \Theta(g(n))$. Pentru toate valorile lui n aflate la dreapta lui n_0, valoarea lui $f(n)$ este mai mare sau egală cu $c_1 g(n)$ şi mai mică sau egală cu $c_2 g(n)$. Cu alte cuvinte, pentru orice $n \ge n_0$, s-ar putea spune că funcţia $f(n)$ este egală cu $g(n)$ până la un factor constant. Vom spune că $g(n)$ este o ***margine asimptotic tare*** pentru $f(n)$.

Definiţia lui $\Theta(g(n))$ cere ca orice membru $f(n) \in \Theta(g(n))$ să fie ***asimptotic nenegativ***, adică $f(n)$ să fie nenegativă pentru n destul de mare. O funcţie asimptotic pozitivă este strict pozitivă pentru orice n suficient de mare. Prin urmare, funcţia $g(n)$ însăşi ar trebui să fie nenegativă, altfel mulţimea $\Theta(g(n))$ este vidă. Din acest motiv vom presupune în continuare că orice funcţie utilizată în cadrul Θ-notaţiei este asimptotic nenegativă. Această presupunere rămâne valabilă şi pentru celelalte notaţii asimptotice definite în acest capitol.

În capitolul 1 am introdus o noţiune neriguroasă de Θ-notaţie care ducea la neglijarea termenilor de ordin inferior şi a coeficientului constant al termenului de ordin maxim. Să justificăm pe scurt această imagine intuitivă, utilizând de data aceasta definiţia, pentru a arăta că $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Pentru a face aceasta, trebuie determinate constantele pozitive c_1, c_2 şi n_0 astfel încât

$$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$$

pentru orice $n \ge n_0$. Împărţind cu n^2, se obţine

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2.$$

Inegalitatea din dreapta va fi verificată pentru orice $n \ge 1$ dacă alegem $c_2 \ge 1/2$. În acelaşi fel, inegalitatea din stânga va fi verificată pentru orice valoare $n \ge 7$ dacă alegem $c_1 \le 1/14$. Astfel, luând $c_1 = 1/14$, $c_2 = 1/2$ şi $n_0 = 7$, se poate verifica faptul că $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Evident, sunt posibile şi alte alegeri pentru aceste constante, însă faptul important aici este doar acela că acestea există. Să mai observăm că aceste constante depind de funcţia $\frac{1}{2}n^2 - 3n$; o altă funcţie aparţinând lui $\Theta(n^2)$ va cere de regulă alte constante.

Putem utiliza definiţia formală şi pentru a verifica faptul că $6n^3 \ne \Theta(n^2)$. Să presupunem, în scopul obţinerii unei contradicţii, că există c_2 şi n_0 astfel încât $6n^3 \le c_2 n^2$ pentru orice $n \ge n_0$. De aici, $n \le c_2/6$, inegalitate care nu poate fi adevărată pentru n oricât de mare deoarece c_2 este o constantă.

Intuitiv, termenii de ordin inferior ai unei funcţii asimptotic pozitive pot fi ignoraţi în determinarea unor margini asimptotic strânse, deoarece aceştia sunt nesemnificativi pentru valori mari ale lui n. O fracţiune infimă a termenului de rang maxim este suficientă pentru a domina termenii de rang inferior. Astfel, dând lui c_1 o valoare ceva mai mică decât coeficientul termenului

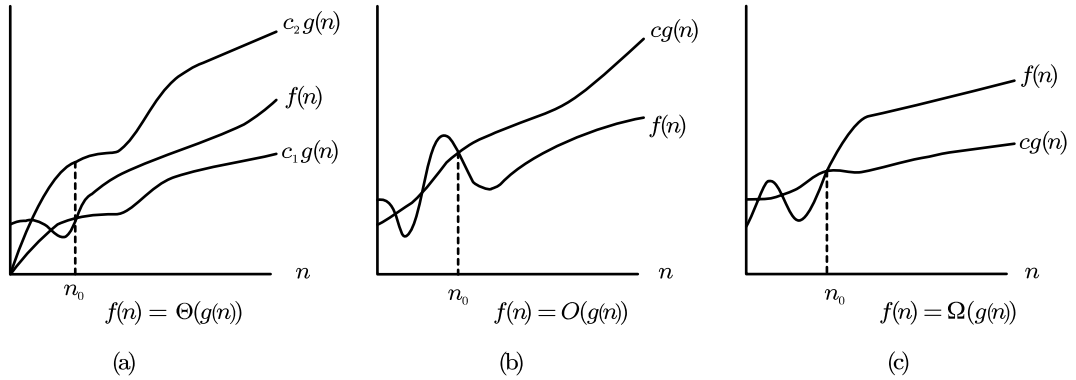


Figura 2.1 Exemple grafice pentru notaţiile Θ, O şi Ω. În fiecare parte, valoarea indicată a lui n_0 este cea mai mică valoare posibilă; orice valoare mai mare va funcţiona, de asemenea. **(a)** Θ-notaţia mărgineşte o funcţie până la factori constanţi. Scriem $f(n) = \Theta(g(n))$ dacă există constantele pozitive n_0, c_1 şi c_2 astfel încât la dreapta lui n_0, valoarea lui $f(n)$ se află întotdeauna între $c_1 g(n)$ şi $c_2 g(n)$ inclusiv. **(b)** O-notaţia dă o margine superioară pentru o funcţie până la un factor constant. Scriem $f(n) = O(g(n))$ dacă există constantele pozitive n_0 şi c astfel încât la dreapta lui n_0, valoarea lui $f(n)$ este întotdeauna mai mică sau egală cu $cg(n)$. **(c)** Ω-notaţia dă o margine inferioară pentru o funcţie până la un factor constant. Scriem $f(n) = \Omega(g(n))$ dacă există constantele pozitive n_0 şi c astfel încât la dreapta lui n_0, valoarea lui $f(n)$ este întotdeauna mai mare sau egală cu $cg(n)$.

de rang maxim şi dând lui c_2 o valoare ceva mai mare, s-ar putea ca inegalităţile din definiţia Θ-notaţiei să fie satisfăcute. Coeficientul termenului de rang maxim poate fi de asemenea ignorat, deoarece acesta poate schimba c_1 şi c_2 doar cu un factor constant.

Pentru a exemplifica, să considerăm o funcţie pătratică oarecare $f(n) = an^2 + bn + c$, în care a, b şi c sunt constante iar $a > 0$. Eliminarea termenilor de rang inferior lui 2 şi neglijarea constantelor conduce la $f(n) = \Theta(n^2)$. Pentru a arăta acelaşi lucru în mod formal, alegem constantele $c_1 = a/4$, $c_2 = 7a/4$ şi $n_0 = 2 \cdot \max\{|b|/a, \sqrt{|c|/a}\}$. Cititorul poate arăta că $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ pentru orice $n \geq n_0$. În general, pentru orice polinom $p(n) = \sum_{i=0}^{d} a_i x^i$, unde a_i sunt constante şi $a_d > 0$, avem $p(n) = \Theta(n^d)$. (vezi problema 2-1)

Deoarece orice constantă este un polinom de gradul 0, putem expima orice funcţie constantă ca $\Theta(n^0)$ sau $\Theta(1)$. Această ultimă notaţie este un mic abuz deoarece nu este clar care variabilă tinde la infinit.[1] Vom folosi des notaţia $\Theta(1)$ înţelegând prin aceasta fie o constantă, fie o funcţie constantă.

O-notaţia

Θ-notaţia delimitează o funcţie asimptotic inferior şi superior. Când avem numai ***o margine asimptotică superioară***, utilizăm O-notaţia. Fiind dată o funcţie $g(n)$, vom nota cu $O(g(n))$,

[1] Adevărata problemă este aceea că notaţia noastră obişnuită pentru funcţii nu face distincţie între funcţii şi valorile lor. În λ-calcul, parametrii unei funcţii sunt clar specificaţi: funcţia n^2 poate fi scrisă ca $\lambda n \cdot n^2$ sau chiar $\lambda r \cdot r^2$. Adoptarea unei notaţii mai riguroase, pe de altă parte, ar complica manipulările algebrice, aşa că preferăm să tolerăm abuzul.

mulţimea

$$O(g(n)) = \{f(n) \quad : \quad \text{există constantele pozitive } c \text{ şi } n_0 \text{ astfel încât} \\ 0 \leq f(n) \leq cg(n) \text{ pentru orice } n \geq n_0\}.$$

Vom utiliza O-notaţia pentru a indica o margine superioară a unei funcţii până la un factor constant. Figura 2.1(b) dă imaginea intuitivă aflată în spatele O-notaţiei. Pentru toate valorile lui n aflate la dreapta lui n_0, valorile corespunzătoare ale lui $f(n)$ sunt egale sau mai mici decât cele ale lui $g(n)$.

Pentru a indica faptul că o funcţie $f(n)$ este un membru al lui $O(g(n))$, vom scrie $f(n) = O(g(n))$. Să notăm faptul că $f(n) = \Theta(g(n))$ implică $f(n) = O(g(n))$ deoarece Θ-notaţia este o noţiune mai puternică decât O-notaţia. În limbajul teoriei mulţimilor, avem $\Theta(g(n)) \subseteq O(g(n))$. Astfel, demonstraţia faptului că orice funcţie pătratică $an^2 + bn + c$, unde $a > 0$, în $\Theta(n^2)$, arată de asemenea că orice funcţie pătratică este în $O(n^2)$. Ceea ce poate părea şi mai surprinzător este că orice funcţie *liniară* $an + b$ este în $O(n^2)$, ceea ce se verifică uşor, luând $c = a + |b|$ şi $n_0 = 1$.

Unor cititori, care s-au mai întâlnit cu notaţii de tipul O, ar putea să li se pară ciudat că scriem $n = O(n^2)$. În literatură, O-notaţia este folosită uneori informal pentru a descrie margini asimptotice strânse, adică ceea ce am definit utilizând Θ-notaţia. În această carte, totuşi, atunci când scriem $f(n) = O(g(n))$, afirmăm, în esenţă, că un anumit multiplu constant al lui $g(n)$ este o margine asimptotică superioară a lui $f(n)$, fără să spunem nimic despre cât de strânsă este această margine superioară. Distincţia dintre marginile asimptotice superioare şi marginile asimptotice strânse a devenit standard în literatura consacrată algoritmilor.

Folosind O-notaţia, se poate adeseori descrie timpul de execuţie al unui algoritm inspectând structura globală a algoritmului. De exemplu, structura de ciclu dublu a algoritmului de sortare prin inserţie din capitolul 1 asigură imediat o margine superioară de tipul $O(n^2)$ pentru timpul de execuţie în cazul cel mai defavorabil: costul ciclului interior este mărginit superior de $O(1)$ (constantă), indicii i, j sunt, ambii, cel mult n iar ciclul interior este executat cel mult o dată pentru fiecare dintre cele n^2 perechi de valori pentru i şi j.

Deoarece O-notaţia descrie o margine superioară, atunci când o folosim pentru delimitarea timpului de execuţie în cazul cel mai defavorabil, prin implicaţie, delimităm, de asemenea, timpul de execuţie al algoritmului pentru date de intrare arbitrare. Astfel, delimitarea $O(n^2)$ pentru timpul de execuţie în cazul cel mai defavorabil al sortării prin inserţie se aplică, în egală măsură, timpului său de execuţie pentru orice date de intrare. Delimitarea $\Theta(n^2)$ a timpului de execuţie în cazul cel mai defavorabil al sortării prin inserţie, totuşi, nu implică o delimitare $\Theta(n^2)$ asupra timpului de execuţie al sortării prin inserţie pentru *orice* date de intrare. De exemplu, am văzut în capitolul 1 că dacă datele de intrare sunt deja sortate, sortarea prin inserţie se execută în timpul $\Theta(n)$.

Tehnic, este o inexactitate să se spună că timpul de execuţie la sortarea prin inserţie este $O(n^2)$ deoarece pentru un n dat timpul real de execuţie depinde de forma datelor de intrare de dimensiune n. Prin urmare, timpul de execuţie nu este de fapt tocmai o funcţie de n. Ceea ce înţelegem atunci când spunem "timpul de execuţie este $O(n^2)$" este că timpul de execuţie în cazul cel mai defavorabil (care este o funcţie de n) este $O(n^2)$, sau, echivalent, oricare ar fi datele de intrare de dimensiune n, pentru fiecare valoare a lui n, timpul de execuţie pentru acest set de date de intrare este $O(n^2)$.

Ω-notaţia

La fel cum O-notaţia furnizează o delimitare asimptotică *superioară* pentru o funcţie, Ω-notaţia furnizează o ***delimitare asimptotică inferioară***. Pentru o funcţie dată $g(n)$, vom nota cu $\Omega(g(n))$, mulţimea

$$\Omega(g(n)) = \{f(n) \ : \ \text{există constantele pozitive } c \text{ şi } n_0 \text{ astfel încât } 0 \leq cg(n) \leq f(n) \text{ pentru orice } n \geq n_0\}$$

Imaginea intuitivă în spatele Ω-notaţiei este arătată în figura 2.1(c). Pentru toate valorile n aflate la dreapta lui n_0, $f(n)$ este mai mare sau egală cu $cg(n)$.

Din definiţiile notaţiilor asimptotice pe care le-am întâlnit până acum, este uşor de demonstrat următoarea teoremă importantă (vezi exerciţiul 2.1-5).

Teorema 2.1 Pentru orice două funcţii $f(n)$ şi $g(n)$, avem $f(n) = \Theta(g(n))$ dacă şi numai dacă $f(n) = O(g(n))$ şi $f(n) = \Omega(g(n))$. ■

Ca un exemplu de aplicare a acestei teoreme, faptul că $an^2 + bn + c = \Theta(n^2)$ implică imediat $an^2 + bn + c = O(n^2)$ şi $an^2 + bn + c = \Omega(n^2)$. În practică, în loc să utilizăm teorema 2.1 pentru a obţine margini asimptotice superioare sau inferioare pornind de la margini asimptotice strânse, aşa cum am făcut în exemplul precedent, o utilizăm de obicei pentru a demonstra delimitări asimptotice strânse plecând de la delimitări asimptotice superioare şi inferioare. Mărginirea asimptotic tare va fi demonstrată utilizând mărginirea asimptotică, superioară şi inferioară.

Deoarece Ω-notaţia descrie o margine inferioară, atunci când o utilizăm pentru a evalua timpul de execuţie al unui algoritm în cazul cel mai favorabil, în mod implicit, evaluăm şi timpul de execuţie al algoritmului pentru date de intrare arbitrare. De exemplu, timpul de execuţie în cazul cel mai favorabil al sortării prin inserţie este $\Omega(n)$, de unde rezultă că timpul de execuţie al sortării prin inserţie este $\Omega(n)$.

Timpul de execuţie al sortării prin inserţie se află prin urmare între $O(n)$ şi $\Omega(n^2)$ deoarece este situat întotdeauna între o funcţie liniară şi o funcţie pătratică de n. Mai mult, aceste aproximări sunt asimptotic cele mai strânse: de exemplu, timpul de execuţie al sortării prin inserţie nu este $\Omega(n^2)$ deoarece acesta este $\Theta(n)$ în cazul datelor de intrare deja sortate. Nu este contradictoriu să spunem totuşi că acest timp de execuţie este $\Omega(n^2)$ *în cazul cel mai defavorabil* deoarece există cel puţin un tip de date de intrare care impun algoritmului un timp de execuţie $\Omega(n^2)$. Atunci când spunem că *timpul de execuţie* al unui algoritm este $\Omega(g(n))$, înţelegem că *nu contează ce set particular de date de intrare de dimensiune* n *este ales pentru fiecare valoare a lui* n, timpul de execuţie pe acel set de date de intrare este cel puţin o constantă înmulţită cu $g(n)$, pentru n suficient de mare.

Notaţia asimptotică în ecuaţii

Am văzut deja cum poate fi utilizată o notaţie asimptotică în formulele matematice. De exemplu, la introducerea O-notaţiei, am scris "$n = O(n^2)$". Am putea scrie, de asemenea $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Cum interpretăm astfel de formule?

Atunci când o notaţie asimptotică se află singură în membrul drept al unei ecuaţii, ca în $n = O(n^2)$, deja am definit semnul de egalitate ca reprezentând apartenenţa din teoria mulţimilor: $n \in O(n^2)$. În general, totuşi, când notaţia asimptotică apare într-o relaţie, o vom interpreta

ca reprezentând o funcţie anonimă pe care nu ne obosim să o numim. De exemplu, egalitatea $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ înseamnă de fapt $2n^2 + 3n + 1 = 2n^2 + f(n)$, unde $f(n)$ este o funcţie oarecare din mulţimea $\Theta(n)$. În acest caz, $f(n) = 3n + 1$, care este într-adevăr în $\Theta(n)$.

Folosirea notaţiilor asimptotice în aceste situaţii poate ajuta la eliminarea unor detalii inutile dintr-o ecuaţie. De exemplu, în capitolul 1, am exprimat timpul de execuţie în cazul cel mai defavorabil la sortarea prin interclasare prin relaţia de recurenţă

$$T(n) = 2\,T(n/2) + \Theta(n)$$

Dacă suntem interesaţi doar de comportarea asimptotică a lui $T(n)$, nu are nici un rost să specificăm exact toţi termenii de ordin inferior: aceştia sunt toţi presupuşi a fi incluşi în funcţia anonimă notată prin termenul $\Theta(n)$.

Numărul funcţiilor "anonime" într-o expresie este înţeles ca fiind egal cu numărul notaţiilor asimptotice care apar în această expresie. De exemplu, în expresia

$$\sum_{i=1}^{n} O(i)$$

există o singură funcţie anonimă (o funcţie de i). Această expresie ***nu*** este echivalentă cu $O(1) + O(2) + \cdots + O(n)$, care oricum nu are o interpretare clară.

În anumite cazuri apar notaţii asimptotice în membrul stâng al unei ecuaţii, cum ar fi

$$2n^2 + \Theta(n) = \Theta(n^2).$$

Vom interpreta astfel de ecuaţii utilizând următoarea regulă: *Oricum ar fi alese funcţiile anonime din partea stângă a semnului egal, există o modalitate de alegere a funcţiilor anonime din partea dreaptă a semnului egal pentru a realiza egalitatea descrisă de ecuaţie.* Astfel, sensul exemplului nostru este acela că pentru ***orice*** funcţie $f(n) \in \Theta(n)$, există ***o anumită*** funcţie $g(n) \in \Theta(n^2)$ astfel încât $2n^2 + f(n) = g(n)$ pentru orice n. Cu alte cuvinte, membrul drept al ecuaţiei oferă un nivel de detaliu mai scăzut decât membrul stâng.

Un număr de astfel de relaţii pot fi înlănţuite ca în

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2)$$

Putem interpreta fiecare ecuaţie separat după regula de mai sus. Prima ecuaţie spune că există o funcţie $f(n) \in \Theta(n)$ astfel încât $2n^2 + 3n + 1 = 2n^2 + f(n)$ pentru orice n. A doua ecuaţie spune că pentru ***orice*** funcţie $g(n) \in \Theta(n)$ (aşa cum este $f(n)$), există o ***anumită*** funcţie $h(n) \in \Theta(n^2)$ astfel încât $2n^2 + g(n) = h(n)$ pentru orice n. Să observăm că această interpretare implică $2n^2 + 3n + 1 = \Theta(n^2)$, fapt care coincide cu informaţia intuitivă pe care o transmite lanţul celor două ecuaţii de mai sus.

o-notaţia

Delimitarea asimptotică superioară dată de O-notaţia definită anterior poate sau nu să fie o delimitare asimptotic strânsă. Astfel, relaţia $2n^2 = O(n^2)$, este o delimitare asimptotic strânsă pe când $2n = O(n^2)$ nu este. Vom utiliza în cele ce urmează o-notaţia pentru a desemna o delimitare superioară care nu este asimptotic strânsă. Formal, vom defini $o(g(n))$ ("o mic de g de n") ca fiind mulţimea

$$o(g(n)) = \{f(n) \quad : \quad \text{pentru orice constantă pozitivă } c > 0 \text{ există o constantă } n_0 > 0 \text{ astfel încât } 0 \leq f(n) < cg(n) \text{ pentru orice } n \geq n_0\}.$$

De exemplu, $2n = o(n^2)$, dar $2n^2 \neq o(n^2)$.

Definiţiile pentru O-notaţie şi o-notaţie sunt similare. Principala diferenţă este aceea că în $f(n) = O(g(n))$, delimitarea $0 \leq f(n) \leq c\,g(n)$ are loc pentru o *anumită* constantă $c > 0$, dar, în $f(n) = o(g(n))$, delimitarea $0 \leq f(n) < c\,g(n)$ are loc pentru *toate* constantele $c > 0$. Intuitiv, în cazul o-notaţiei, funcţia $f(n)$ devine neglijabilă relativ la $g(n)$ atunci când n tinde la infinit; cu alte cuvinte

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0. \tag{2.1}$$

Unii autori folosesc această limită ca o definiţie pentru o-notaţie: definiţia din această carte impune, de asemenea, funcţiilor anonime să fie asimptotic nenegative.

ω-notaţia

Prin analogie, ω-notaţia este faţă de Ω-notaţie ceea ce este o-notaţia faţă de O-notaţie. Utilizăm ω-notaţia pentru a desemna o delimitare asimptotică inferioară care nu este asimptotic strânsă. O cale de a o defini este

$f(n) \in \omega(g(n))$ dacă şi numai dacă $g(n) \in o(f(n))$.

Formal, totuşi, definim $\omega(g(n))$ ("omega mic de g de n") ca fiind mulţimea

$$\begin{aligned}\omega(g(n)) = \{f(n) \quad : \quad & \text{pentru orice constantă pozitivă } c > 0 \text{ există o constantă} \\ & n_0 > 0 \text{ astfel încât } 0 \leq cg(n) < f(n) \text{ pentru orice } n \geq n_0\}.\end{aligned}$$

De exemplu, $n^2/2 = \omega(n)$, dar $n^2/2 \neq \omega(n)$. Relaţia $f(n) = \omega(g(n))$ implică

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty,$$

dacă limita există. Cu alte cuvinte, $f(n)$ devine oricât de mare relativ la $g(n)$ atunci când n tinde la infinit.

Compararea funcţiilor

Multe dintre proprietăţile relaţiilor dintre numerele reale se aplică şi la compararea asimptotică a funcţiilor. Pentru cele ce urmează vom presupune că $f(n)$ şi $g(n)$ sunt asimptotic pozitive.

Tranzitivitatea:

$$\begin{array}{llllll}
f(n) = \Theta(g(n)) & \text{şi} & g(n) = \Theta(h(n)) & \text{implică} & f(n) = \Theta(h(n)), \\
f(n) = O(g(n)) & \text{şi} & g(n) = O(h(n)) & \text{implică} & f(n) = O(h(n)), \\
f(n) = \Omega(g(n)) & \text{şi} & g(n) = \Omega(h(n)) & \text{implică} & f(n) = \Omega(h(n)), \\
f(n) = o(g(n)) & \text{şi} & g(n) = o(h(n)) & \text{implică} & f(n) = o(h(n)), \\
f(n) = \omega(g(n)) & \text{şi} & g(n) = \omega(h(n)) & \text{implică} & f(n) = \omega(h(n)).
\end{array}$$

Reflexivitatea:

$$\begin{aligned}
f(n) &= \Theta(f(n)), \\
f(n) &= O(f(n)), \\
f(n) &= \Omega(f(n)).
\end{aligned}$$

Simetria:

$$f(n) = \Theta(g(n)) \quad \text{dacă şi numai dacă} \quad g(n) = \Theta(f(n)).$$

Antisimetria:

$$\begin{array}{ll} f(n) = O(g(n)) & \text{dacă şi numai dacă} \quad g(n) = \Omega(f(n)), \\ f(n) = o(g(n)) & \text{dacă şi numai dacă} \quad g(n) = \omega(f(n)). \end{array}$$

Deoarece aceste proprietăţi sunt valide pentru notaţii asimptotice, se poate trasa o analogie între compararea asimptotică a două funcţii f şi g şi compararea asimptotică a două numere reale a şi b:

$$\begin{array}{lcl} f(n) = O(g(n)) & \approx & a \leq b, \\ f(n) = \Omega(g(n)) & \approx & a \geq b, \\ f(n) = \Theta(g(n)) & \approx & a = b, \\ f(n) = o(g(n)) & \approx & a < b, \\ f(n) = \omega(g(n)) & \approx & a > b. \end{array}$$

O proprietate a numerelor reale, totuşi, nu se transpune la notaţii asimptotice:

Trihotomia: Pentru orice două numere reale a şi b exact una dintre următoarele relaţii este adevărată: $a < b$, $a = b$ sau $a > b$.

Deşi orice două numere reale pot fi comparate, nu toate funcţiile sunt asimptotic comparabile. Cu alte cuvinte, pentru două funcţii $f(n)$ şi $g(n)$, se poate întâmpla să nu aibă loc nici $f(n) = O(g(n))$, nici $f(n) = \Omega(g(n))$. De exemplu, funcţiile n şi $n^{1+\sin n}$ nu pot fi comparate utilizând notaţii asimptotice, deoarece valoarea exponentului în $n^{1+\sin n}$ oscilează între 0 şi 2, luând toate valorile intermediare.

Exerciţii

2.1-1 Fie $f(n)$ şi $g(n)$ funcţii asimptotic nenegative. Utilizând definiţia de bază a Θ-notaţiei, demonstraţi că $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

2.1-2 Demonstraţi că pentru orice constante reale a şi b, unde $b > 0$, avem

$$(n + a)^b = \Theta(n^b). \tag{2.2}$$

2.1-3 Explicaţi de ce afirmaţia "Timpul de execuţie al algoritmului A este cel puţin $O(n^2)$" este lipsită de conţinut.

2.1-4 Este adevărat că $2^{n+1} = O(2^n)$? Este adevărat că $2^{2n} = O(2^n)$?

2.1-5 Demonstraţi teorema 2.1.

2.1-6 Demonstraţi că timpul de execuţie al unui algoritm este $\Theta(g(n))$ dacă şi numai dacă timpul său de execuţie în cazul cel mai defavorabil este $O(g(n))$ şi timpul său de execuţie în cazul cel mai favorabil este $\Omega(g(n))$.

2.1-7 Demonstraţi că $o(g(n)) \cap \omega(g(n))$ este mulţimea vidă.

2.1-8 Putem extinde notaţia noastră la cazul a doi parametri n şi m care tind la infinit în mod independent, cu viteze diferite. Peentru o funcţie dată $g(n,m)$ notăm cu $O(g(n,m))$ mulţimea de funcţii

$$O(g(n,m)) = \{f(n,m) \quad : \quad \text{există constantele pozitive } c, n_0 \text{ şi } m_0 \text{ astfel încât} \\ 0 \le f(n,m) \le cg(n,m) \text{ pentru orice } n \ge n_0 \text{ şi } m \ge m_0\}.$$

Daţi definiţiile corespunzătoare pentru $\Omega(g(n,m))$ şi $\Theta(g(n,m))$.

2.2. Notaţii standard şi funcţii comune

Această secţiune trece în revistă câteva funcţii şi notaţii matematice standard şi explorează relaţiile dintre ele. Ea ilustrează, de asemenea, utilizarea notaţiilor asimptotice.

Monotonie

O funcţie $f(n)$ este ***monoton crescătoare*** dacă $m \le n$ implică $f(m) \le f(n)$. Analog, ea este ***monoton descrescătoare*** dacă $m \le n$ implică $f(m) \ge f(n)$. O funcţie $f(n)$ este ***strict crescătoare*** dacă $m < n$ implică $f(m) < f(n)$ şi ***strict descrescătoare*** dacă $m < n$ implică $f(m) > f(n)$.

Părţi întregi inferioare şi superioare

Pentru orice număr real x, notăm cel mai mare întreg mai mic sau egal cu x prin $\lfloor x \rfloor$ (se citeşte "partea întreagă inferioară a lui x") şi cel mai mic întreg mai mare sau egal cu x prin $\lceil x \rceil$ (se citeşte "partea întreagă superioară a lui x"). Pentru orice x real,

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1.$$

Pentru orice întreg n,

$$\lceil n/2 \rceil + \lfloor n/2 \rfloor = n,$$

iar pentru orice întreg n şi orice întregi $a \ne 0$ şi $b > 0$,

$$\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil \tag{2.3}$$

şi

$$\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor. \tag{2.4}$$

Funcţiile parte întreagă inferioară şi superioară sunt monoton crescătoare.

Polinoame

Fiind dat un întreg pozitiv d, ***un polinom în*** n ***de gradul*** d este o funcţie $p(n)$ de forma

$$p(n) = \sum_{i=0}^{d} a_i n^i,$$

unde constantele $a_0, a_1, \ldots a_d$ sunt ***coeficienţii*** polinomului, iar $a_d \neq 0$. Un polinom este *asimptotic pozitiv* dacă şi numai dacă $a_d > 0$. Pentru un polinom asimptotic pozitiv $p(n)$ de grad d avem $p(n) = \Theta(n^d)$. Pentru orice constantă reală $a \geq 0$, funcţia n^a este monoton crescătoare, iar pentru orice constantă reală $a \leq 0$ funcţia n^a este monoton descrescătoare. Spunem că o funcţie $f(n)$ este ***polinomial mărginită*** dacă $f(n) = n^{O(1)}$, ceea ce este echivalent cu a spune că $f(n) = O(n^k)$ pentru o anumită constantă k (vezi exerciţiul 2.2-2).

Exponenţiale

Pentru orice $a \neq 0$, m şi n reale avem următoarele identităţi:

$$\begin{aligned} a^0 &= 1, \\ a^1 &= a, \\ a^{-1} &= 1/a, \\ (a^m)^n &= a^{mn}, \\ (a^m)^n &= (a^n)^m, \\ a^m a^n &= a^{m+n}. \end{aligned}$$

Pentru orice n şi $a \geq 1$, funcţia a^n este monoton crescătoare în n. Când ne va conveni, vom presupune că $0^0 = 1$.

Ratele de creştere ale polinoamelor şi exponenţialelor pot fi legate prin următorul fapt. Pentru orice constante a şi b astfel încât $a > 1$,

$$\lim_{n \to \infty} \frac{n^b}{a^n} = 0, \tag{2.5}$$

din care putem trage concluzia că

$$n^b = o(a^n).$$

Astfel, orice funcţie exponenţială având baza strict mai mare decât 1 creşte mai repede decât orice funcţie polinomială.

Utilizând e pentru a nota 2.71828..., baza funcţiei logaritm natural, avem, pentru orice x real,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}, \tag{2.6}$$

unde "!" desemnează funcţia factorial definită mai târziu în această secţiune. Pentru orice x real avem inegalitatea

$$e^x \geq 1 + x, \tag{2.7}$$

unde egalitatea are loc numai pentru $x = 0$. Când $|x| \leq 1$, avem aproximaţia

$$1 + x \leq e^x \leq 1 + x + x^2. \tag{2.8}$$

Când $x \to 0$, aproximaţia lui e^x prin $1 + x$ este destul de bună:

$$e^x = 1 + x + \Theta(x^2).$$

(În această ecuaţie, notaţia asimptotică este utilizată pentru a descrie comportamentul la limită pentru $x \to 0$ mai degrabă decât pentru $x \to \infty$.) Avem, pentru orice x,

$$\lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n = e^x.$$

Logaritmi

Vom utiliza următoarele notaţii:

$$\begin{array}{rcll} \lg n & = & \log_2 n & \text{(logaritm binar)}, \\ \ln n & = & \log_e n & \text{(logaritm natural)}, \\ \lg^k n & = & (\lg n)^k & \text{(exponenţierea)}, \\ \lg\lg n & = & \lg(\lg n) & \text{(compunerea)}. \end{array}$$

O convenţie de notaţie importantă pe care o vom adopta este aceea că ***funcţiile logaritmice se vor aplica numai următorului termen dintr-o formulă***, astfel că $\lg n + k$ va însemna $(\lg n) + k$ şi nu $\lg(n + k)$. Pentru $n > 0$ şi $b > 1$, funcţia $\log_b n$ este strict crescătoare.

Pentru orice numere reale $a > 0, b > 0, c > 0$ şi n,

$$\begin{array}{rcl} a & = & b^{\log_b a}, \\ \log_c(ab) & = & \log_c a + \log_c b, \\ \log_b a^n & = & n \log_b a, \\ \log_b a & = & \dfrac{\log_c a}{\log_c b}, \\ \log_b(1/a) & = & -\log_b a, \\ \log_b a & = & \dfrac{1}{\log_a b}, \\ a^{\log_b n} & = & n^{\log_b a}. \end{array} \tag{2.9}$$

Deoarece schimbarea bazei unui logaritm de la o constantă la o altă constantă schimbă valoarea logaritmului doar printr-un factor constant, vom utiliza adesea notaţia "$\lg n$" când factorii constanţi nu vor fi importanţi, aşa cum este cazul cu O-notaţia. Informaticienilor li se pare că 2 este baza cea mai naturală pentru logaritmi, deoarece atât de mulţi algoritmi şi structuri de date presupun descompunerea unei probleme în două părţi.

Există o dezvoltare în serie simplă pentru $\ln(1 + x)$ când $|x| < 1$:

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} - \cdots$$

Avem, de asemenea, următoarele inegalităţi pentru $x > -1$:

$$\frac{x}{1 + x} \leq \ln(1 + x) \leq x, \tag{2.10}$$

unde egalitatea are loc numai pentru $x = 0$.

Spunem că o funcţie $f(n)$ este ***polilogaritmic mărginită*** dacă $f(n) = \lg^{O(1)} n$. Putem lega creşterile polinoamelor şi polilogaritmilor înlocuind pe n cu $\lg n$ şi pe a cu 2^a în ecuaţia (2.5), obţinând

$$\lim_{n\to\infty} \frac{\lg^b n}{(2^a)^{\lg n}} = \lim_{n\to\infty} \frac{\lg^b n}{n^a} = 0.$$

Din această limită, putem conchide că

$$\lg^b n = o(n^a)$$

pentru orice constantă $a > 0$. Astfel, orice funcţie polinomială pozitivă creşte mai repede decât orice funcţie polilogaritmică.

Factoriali

Notaţia $n!$ (se citeşte "n factorial") este definită pentru numere întregi $n \geq 0$ prin

$$n! = \begin{cases} 1 & \text{dacă } n = 0, \\ n \cdot (n-1)! & \text{dacă } n > 0. \end{cases}$$

Astfel, $n! = 1 \cdot 2 \cdot 3 \cdots n$.

O margine superioară slabă a funcţiei factorial este $n! \leq n^n$, deoarece fiecare dintre cei n factori ai produsului factorial este cel mult n. ***Aproximaţia lui Stirling,***

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right), \tag{2.11}$$

unde e este baza logaritmilor naturali, ne dă o margine superioară mai strânsă, dar şi o margine inferioară. Utilizând aproximaţia lui Stirling, putem demonstra că

$$\begin{aligned} n! &= o(n^n), \\ n! &= \omega(2^n), \\ \lg(n!) &= \Theta(n \lg n). \end{aligned}$$

Următoarele delimitări sunt, de asemenea, valabile pentru orice n:

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n}\left(\frac{n}{e}\right)^{ne^{1/12n}} \tag{2.12}$$

Funcţia logaritm iterată

Utilizăm notaţia $\lg^* n$ (se citeşte "logaritm stea de n") pentru a nota logaritmul iterat, care este definit după cum urmează. Fie funcţia $\lg^{(i)} n$ definită recursiv pentru întregii nenegativi i prin

$$lg^{(i)} n = \begin{cases} n & \text{dacă } i = 0, \\ lg(\lg^{(i-1)} n) & \text{dacă } i > 0 \text{ şi } \lg^{(i-1)} n > 0, \\ \text{nedefinită} & \text{dacă } i > 0 \text{ şi } lg^{(i-1)} n \geq 0 \text{ sau } \lg^{(i-1)} n \text{ este nedefinită.} \end{cases}$$

Calculaţi diferenţa între funcţia $\lg^{(i)} n$ (funcţia logaritm aplicată succesiv de i ori, începând cu argumentul n) şi $\lg^i n$ (logaritmul lui n ridicat la puterea i). Funcţia logaritm iterată este definită prin

$$\lg^* n = \min \left\{ i \geq 0 : \lg^{(i)} n \leq 1 \right\}.$$

Funcţia logaritm iterată este o funcţie *foarte* lent crescătoare:

$$\begin{aligned} \lg^* 2 &= 1, \\ \lg^* 4 &= 2, \\ \lg^* 16 &= 3, \\ \lg^* 65536 &= 4, \\ \lg^*(2^{65536}) &= 5. \end{aligned}$$

Deoarece numărul atomilor din universul observabil este estimat la aproximativ 10^{80}, care este mult mai mic decât 2^{65536}, rareori întâlnim o valoare a lui n astfel încât $\lg^* n > 5$.

Numere Fibonacci

Numerele Fibonacci sunt definite prin următoarea recurenţă:

$$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2} \quad \text{pentru } i \geq 2. \tag{2.13}$$

Astfel, fiecare număr Fibonacci este suma celor două numere Fibonacci anterioare, rezultând secvenţa

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots .$$

Numerele Fibonacci sunt legate de ***raportul de aur*** ϕ şi de conjugatul său $\widehat{\phi}$, care sunt date de următoarele formule:

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots, \widehat{\phi} = \frac{1-\sqrt{5}}{2} = -.61803\ldots. \tag{2.14}$$

Mai precis, avem

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}}, \tag{2.15}$$

ceea ce se poate demonstra prin inducţie (exerciţiul 2.2-7). Deoarece $|\widehat{\phi}| < 1$, avem $|\widehat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$, astfel că cel de-al i-lea număr Fibonacci este egal cu $\phi^i/\sqrt{5}$ rotunjit la cel mai apropiat întreg. Astfel, numerele Fibonacci cresc exponenţial.

Exerciţii

2.2-1 Arătaţi că dacă $f(n)$ şi $g(n)$ sunt funcţii monoton crescătoare, atunci la fel sunt şi funcţiile $f(n) + g(n)$ şi $f(g(n))$, iar dacă $f(n)$ şi $g(n)$ sunt în plus nenegative, atunci şi $f(n) \cdot g(n)$ este monoton crescătoare.

2.2-2 Utilizaţi definiţia O-notaţiei pentru a arăta că $T(n) = n^{O(1)}$ dacă şi numai dacă există o constantă $k > 0$ astfel încât $T(n) = O(n^k)$.

2.2-3 Demonstraţi ecuaţia (2.9).

2.2-4 Demonstraţi că $\lg(n!) = \Theta(n \lg n)$ şi că $n! = o(n^n)$.

2.2-5 $\star$ Este funcţia $\lceil \lg n \rceil !$ polinomial mărginită? Este funcţia $\lceil \lg \lg n \rceil !$ polinomial mărginită?

2.2-6 $\star$ Care este asimptotic mai mare: $\lg(\lg^* n)$ sau $\lg^*(\lg n)$?

2.2-7 Demonstraţi prin inducţie că cel de-al i-lea număr Fibonacci verifică egalitatea $F_i = (\phi^i - \widehat{\phi}^i)/\sqrt{5}$, unde ϕ este raportul de aur iar $\widehat{\phi}$ este conjugatul său.

2.2-8 Demonstraţi că pentru $i \geq 0$ cel de-al $(i+2)$-lea număr Fibonacci verifică $F_{i+2} \geq \phi^i$.

Probleme

2-1 Comportamentul asimptotic al polinoamelor
Fie

$$p(n) = \sum_{i=0}^{d} a_i p^i,$$

unde $a_d > 0$ un polinom de gradul d în n şi fie k o constantă. Utilizaţi proprietăţile notaţiilor asimptotice pentru a demonstra următoarele proprietăţi.

a. Dacă $k \geq d$, atunci $p(n) = O(n^k)$.

b. Dacă $k \leq d$, atunci $p(n) = \Omega(n^k)$.

c. Dacă $k = d$, atunci $p(n) = \Theta(n^k)$.

d. Dacă $k > d$, atunci $p(n) = o(n^k)$.

e. Dacă $k < d$, atunci $p(n) = \omega(n^k)$.

2-2 Creşteri asimptotice relative
Indicaţi, pentru fiecare pereche de expresii (A, B) din tabelul următor, dacă A este O, o, Ω, ω sau Θ de B. Presupuneţi că $k \geq 1$, $\epsilon > 0$ şi $c > 1$ sunt constante. Răspunsul trebuie să fie sub forma unui tabel, cu "da" sau "nu" scrise în fiecare rubrică.

	A	B	O	o	Ω	ω	Θ
a.	$\lg^k n$	n^ϵ					
b.	n^k	c^n					
c.	$\sqrt{n}$	$n^{\sin n}$					
d.	2^n	$2^{n/2}$					
e.	$n^{\lg m}$	$m^{\lg n}$					
f.	$\lg(n!)$	$\lg(n^n)$					

2-3 Ordonarea după ratele de creştere asimptotică

a. Ordonaţi următoarele funcţii după ordinea de creştere; mai precis, găsiţi un aranjament de funcţii $g_1, g_2, \ldots g_{30}$ care să verifice $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$. Partiţionaţi lista în clase de echivalenţă astfel încât $f(n)$ şi $g(n)$ să fie în aceeaşi clasă de echivalenţă dacă şi numai dacă $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$\left(\frac{3}{2}\right)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln\ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg\lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2\lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

b. Daţi un exemplu de o singură funcţie nenegativă $f(n)$ astfel încât pentru toate funcţiile $g_i(n)$ de la punctul (a), $f(n)$ să nu fie nici $O(g_i(n))$ nici $\Omega(g_i(n))$.

2-4 Proprietăţi ale notaţiilor asimptotice

Fie $f(n)$ şi $g(n)$ funcţii asimptotic pozitive. Demonstraţi următoarele conjecturi sau demonstraţi că ele nu sunt adevărate.

a. $f(n) = O(g(n)$ implică $g(n) = O(f(n)$.

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.

c. $f(n) = O(g(n))$ implică $\lg(f(n)) = O(\lg(g(n)))$, unde $\lg(g(n)) > 0$ şi $f(n) \geq 1$ pentru orice n suficient de mare.

d. $f(n) = O(g(n))$ implică $2^{f(n)} = O(2^{g(n)})$.

e. $f(n) = O((f(n))^2)$.

f. $f(n) = O(g(n))$ implică $g(n) = \Omega(f(n))$.

g. $f(n) = \Theta(f(n/2))$.

h. $f(n) + o(f(n)) = \Theta(f(n))$.

2-5 Variaţii ale lui O şi Ω

Unii autori definesc Ω într-un mod uşor diferit de al nostru; să utilizăm $\overset{\infty}{\Omega}$ (se citeşte "omega infinit") pentru această definiţie alternativă. Spunem că $f(n) = \overset{\infty}{\Omega}(g(n))$ dacă există o constantă pozitivă c astfel încât $f(n) \geq cg(n) \geq 0$ pentru infinit de mulţi n.

a. Arătaţi că pentru oricare două funcţii $f(n)$ şi $g(n)$ care sunt asimptotic nenegative, fie $f(n) = O(g(n))$, fie $f(n) = \overset{\infty}{\Omega}(g(n))$ fie amândouă, în timp ce această afirmaţie nu este adevărată dacă utilizăm Ω în loc de $\overset{\infty}{\Omega}$.

b. Descrieţi avantajele şi dezavantajele potenţiale ale utilizării lui $\overset{\infty}{\Omega}$ în locul lui Ω pentru a descrie timpii de execuţie ai programelor.

Unii autori îl definesc şi pe O într-un mod uşor diferit; să utilizăm O' pentru definiţia alternativă. Spunem că $f(n) = O'(g(n))$ dacă şi numai dacă $|f(n)| = O(g(n))$.

c. Ce se întâmplă cu fiecare dintre direcţiile lui "dacă şi numai dacă" din teorema 2.1 cu această nouă definiţie?

Unii autori definesc $\tilde{O}$ (se citeşte "o moale") ca însemnând O cu factorii logaritmici ignoraţi:

$$\tilde{O}(g(n)) = \{f(n) \quad : \quad \text{există constantele pozitive, } c, k \text{ şi } n_0 \text{ astfel încât} \\ 0 \le f(n) \le cg(n)\lg^k(n) \text{ pentru orice } n \ge n_0\}.$$

d. Definiţi $\tilde{\Omega}$ şi $\tilde{\Theta}$ într-un mod analog. Demonstraţi proprietatea analogă corespunzătoare teoremei 2.1.

2-6 Funcţii iterate

Operatorul de iteraţie "*" utilizat în funcţia $\lg^*$ poate fi aplicat funcţiilor monoton crescătoare pe mulţimea numerelor reale. Pentru o funcţie f care satisface $f(n) < n$ definim funcţia $f^{(i)}$ recursiv pentru numerele întregi i nenegative prin

$$f^{(i)}(n) = \begin{cases} f(f^{(i-1)}(n)) & \text{dacă } i > 0, \\ n & \text{dacă } i = 0. \end{cases}$$

Pentru o constantă dată $c \in \mathbf{R}$, definim funcţia iterată f_c^* prin

$$f_c^*(n) = \min\{i \ge 0 : f^{(i)}(n) \le c\},$$

care nu trebuie să fie bine-definită în toate cazurile. Cu alte cuvinte, cantitatea $f_c^*(n)$ este numărul de aplicaţii iterate ale funcţiei f necesare pentru a reduce argumentul la c sau la mai puţin.

Pentru fiecare dintre următoarele funcţii $f(n)$ şi constantele c, daţi o evaluare cât se poate de strânsă a lui $f_c^*(n)$.

	$f(n)$	c	$f_c^*(n)$
a.	$\lg n$	1	
b.	$n-1$	0	
c.	$n/2$	1	
d.	$n/2$	2	
e.	$\sqrt{n}$	2	
f.	$\sqrt{n}$	1	
g.	$n^{1/3}$	2	
h.	$n/\lg n$	2	

Note bibliografice

Knuth [121] investighează originea O-notaţiei şi afirmă că ea apare pentru prima dată într-un text de teoria numerelor al lui P. Bachmann din 1982. o-notaţia a fost inventată de către E. Landau în 1909 pentru discuţia distribuţiei numerelor prime. Notaţiile Ω şi Θ au fost introduse de către Knuth [124] pentru a corecta practica din literatură, populară dar derutantă din punct de vedere tehnic, de a utiliza O-notaţia atât pentru margini superioare cât şi pentru margini

inferioare. Multă lume continuă să utilizeze O-notaţia acolo unde Θ-notaţia este mai precisă din punct de vedere tehnic. O altă discuţie pe tema istoriei şi dezvoltării notaţiilor asimptotice poate fi găsită în Knuth [121, 124] şi Brassard şi Bratley [33].

Nu toţi autorii definesc notaţiile asimptotice în acelaşi mod, deşi diferitele definiţii concordă în cele mai comune situaţii. Unele dintre definiţiile alternative se aplică şi la funcţii care nu sunt asimptotic nenegative, dacă valorile lor absolute sunt mărginite în mod corespunzător.

Alte proprietăţi ale funcţiilor matematice elementare pot fi găsite în orice carte de referinţă de matematică, cum ar fi Abramowitz şi Stegun [1] sau Beyer [27], sau într-o carte de analiză, cum ar fi Apostol [12] sau Thomas şi Finney [192]. Knuth [121] conţine o mulţime de materiale despre matematica discretă, utilizată în informatică.

3 Sume

Când un algoritm conţine o structură de control iterativă cum ar fi un ciclu **cât timp** sau **pentru**, timpul său de execuţie poate fi exprimat ca suma timpilor necesari la fiecare execuţie a corpului ciclului. De exemplu, am văzut în secţiunea 1.2 că a j-a iteraţie a sortării prin inserţie a necesitat un timp proporţional cu j în cazul cel mai defavorabil. Adunând timpii necesari pentru fiecare iteraţie, am obţinut o sumă (sau serie)

$$\sum_{j=2}^{n} j.$$

Evaluarea sumei a dat o margine $\Theta(n^2)$ pentru timpul de execuţie al algoritmului în cazul cel mai defavorabil. Acest exemplu indică importanţa generală a înţelegerii manipulării şi delimitării sumelor. (După cum vom vedea în capitolul 4, sumele apar şi când utilizăm anumite metode pentru rezolvarea recurenţelor.)

Secţiunea 3.1 prezintă mai multe formule de bază în care intervin sume. Secţiunea 3.2 oferă tehnici utile pentru delimitarea sumelor. Formulele din secţiunea 3.1 sunt date fără demonstraţie, deşi demonstraţiile pentru unele dintre ele sunt prezentate în secţiunea 3.2, pentru a ilustra metodele acelei secţiuni. Majoritatea celorlalte demonstraţii pot fi găsite în orice manual de analiză.

3.1. Formule de însumare şi proprietăţi

Fiind dat un şir de numere $a_1, a_2, \ldots$, suma finită $a_1 + a_2 + \ldots + a_n$ poate fi scrisă

$$\sum_{k=1}^{n} a_k.$$

Dacă $n = 0$, valoarea sumei este definită ca fiind 0. Dacă n nu este un număr întreg, mai presupunem că limita superioară este $\lfloor n \rfloor$. Analog, dacă suma începe cu $k = x$, unde x nu este un întreg, presupunem că valoarea iniţială pentru însumare este $\lfloor x \rfloor$. (În general, vom scrie în mod explicit părţile întregi inferioare sau superioare). Valoarea unei serii finite este întotdeauna bine definită, iar termenii săi pot fi adunaţi în orice ordine.

Fiind dat un şir de numere $a_1, a_2, \ldots$, suma infinită $a_1 + a_2 + \ldots$ poate fi scrisă

$$\sum_{k=1}^{\infty} a_k,$$

care este interpretată ca

$$\lim_{n \to \infty} \sum_{k=1}^{n} a_k.$$

Dacă limita nu există, seria ***diverge***; altfel, ea ***converge***. Termenii unei serii convergente nu pot fi întotdeauna adunaţi în orice ordine. Putem rearanja, totuşi, termenii unei ***serii absolut convergente***, care este o serie $\sum_{k=1}^{\infty} a_k$ pentru care seria $\sum_{k=1}^{\infty} |a_k|$ converge de asemenea.

Liniaritate

Pentru orice număr real c şi orice şiruri finite $a_1, a_2, \ldots, a_n$ şi $b_1, b_2, \ldots, b_n$

$$\sum_{k=1}^{n}(ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b_k.$$

Proprietatea de liniaritate este verificată, de asemenea, şi de seriile infinite convergente.

Proprietatea de liniaritate poate fi exploatată pentru a manipula sumele care încorporează notaţii asimptotice. De exemplu,

$$\sum_{k=1}^{n}\Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right).$$

În această ecuaţie, Θ-notaţia din membrul stâng se aplică variabilei k, dar în membrul drept ea se aplică lui n. Astfel de manipulări pot fi aplicate şi seriilor infinite convergente.

Serii aritmetice

Suma

$$\sum_{k=1}^{n} k = 1 + 2 + \ldots + n,$$

care a apărut când am analizat sortarea prin inserţie, este o ***serie aritmetică*** şi are valoarea

$$\begin{aligned}\sum_{k=1}^{n} k &= \frac{1}{2}n(n+1) &(3.1)\\ &= \Theta(n^2) &(3.2)\end{aligned}$$

Serii geometrice

Pentru orice $x \neq 1$ real, suma

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n$$

este o ***serie geometrică*** sau ***exponenţială*** şi are valoarea

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}. \tag{3.3}$$

Când suma este infinită şi $|x| < 1$ avem seria geometrică infinită descrescătoare

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}. \tag{3.4}$$

Serii armonice

Pentru întregi pozitivi n, al n-lea ***număr armonic*** este

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1). \tag{3.5}$$

Integrarea şi diferenţierea seriilor

Se pot obţine formule suplimentare prin integrarea sau diferenţierea formulelor de mai sus. De exemplu, diferenţiind ambii membri ai seriei geometrice infinite (3.4) şi înmulţind cu x, obţinem

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \tag{3.6}$$

Serii telescopante

Pentru orice şir $a_0, a_1, \ldots, a_n$

$$\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0 \tag{3.7}$$

deoarece fiecare dintre termenii $a_1, a_2, \ldots, a_{n-1}$ este adunat exact o dată şi scăzut exact o dată. Spunem că suma ***telescopează***. Analog,

$$\sum_{k=0}^{n-1} (a_k - a_{k+1}) = a_0 - a_n.$$

Ca un exemplu de sumă telescopantă, să considerăm seria

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)}.$$

Deoarece putem rescrie fiecare termen ca

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1},$$

obţinem

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left(\frac{1}{k} - \frac{1}{k+1} \right) = 1 - \frac{1}{n}.$$

Produse

Produsul finit $a_1 a_2 \cdots a_n$ poate fi scris

$$\prod_{k=1}^{n} a_k.$$

Dacă $n = 0$, valoarea produsului este definită ca fiind 1. Putem converti o formulă cu un produs într-o formulă cu o sumă, utilizând identitatea

$$\lg\left(\prod_{k=1}^{n} a_k\right) = \sum_{k=1}^{n} \lg a_k.$$

Exerciţii

3.1-1 Găsiţi o formulă simplă pentru $\sum_{k=1}^{n}(2k-1)$.

3.1-2 ⋆ Arătaţi că $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$ manipulând seria armonică.

3.1-3 ⋆ Arătaţi că $\sum_{k=0}^{\infty}(k-1)/2^k = 0$.

3.1-4 ⋆ Evaluaţi suma $\sum_{k=1}^{\infty}(2k+1)x^{2k}$.

3.1-5 Utilizaţi proprietatea de liniaritate a însumării pentru a demonstra că $\sum_{k=1}^{n} O(f_k(n)) = O\left(\sum_{k=1}^{n} f_k(n)\right)$.

3.1-6 Demonstraţi că $\sum_{k=1}^{\infty} \Omega(f(k)) = \Omega\left(\sum_{k=1}^{\infty} f(k)\right)$.

3.1-7 Evaluaţi produsul $\prod_{k=1}^{n} 2 \cdot 4^k$.

3.1-8 ⋆ Evaluaţi produsul $\prod_{k=2}^{n}(1 - 1/k^2)$.

3.2. Delimitarea sumelor

Există multe tehnici disponibile pentru delimitarea sumelor care descriu timpii de execuţie ai algoritmilor. Iată câteva dintre metodele cel mai frecvent utilizate.

Inducţia matematică

Cea mai simplă cale de a evalua o serie este utilizarea inducţiei matematice. De exemplu, să demonstrăm că seria aritmetică $\sum_{k=1}^{n} k$ are valoarea $\frac{1}{2}n(n+1)$. Putem verifica uşor pentru $n = 1$, aşa că facem ipoteza de inducţie că formula este adevărată pentru orice n şi demonstrăm că are loc pentru $n + 1$. Avem

$$\sum_{k=1}^{n+1} k = \sum_{k=1}^{n} k + (n+1) = \frac{1}{2}n(n+1) + (n+1) = \frac{1}{2}(n+1)(n+2).$$

Nu trebuie să ghicim valoarea exactă a unei sume pentru a utiliza inducţia matematică. Inducţia poate fi utilizată şi pentru a demonstra o delimitare. De exemplu, să demonstrăm că seria geometrică $\sum_{k=0}^{n} 3^k$ este $O(3^n)$. Mai precis, să demonstrăm că $\sum_{k=0}^{n} 3^k \leq c \cdot 3^n$ pentru o anumită constantă c. Pentru condiţia iniţială $n = 0$ avem $\sum_{k=0}^{0} 3^k = 1 \leq c \cdot 1$, cât timp $c \geq 1$. Presupunând că delimitarea are loc pentru n, să demonstrăm că are loc pentru $n + 1$. Avem

$$\sum_{k=0}^{n+1} 3^k = \sum_{k=0}^{n} 3^k + 3^{n+1} \leq c \cdot 3^n + 3^{n+1} = \left(\frac{1}{3} + \frac{1}{c}\right) c \cdot 3^{n+1} \leq c \cdot 3^{n+1}$$

cât timp $(1/3 + 1/c) \leq 1$ sau, echivalent, $c \geq 3/2$. Astfel, $\sum_{k=0}^{n} 3^k = O(3^n)$, ceea ce am dorit să arătăm.

Trebuie să fim prudenţi când utilizăm notaţii asimptotice pentru a demonstra delimitări prin inducţie. Considerăm următoarea demonstraţie vicioasă a faptului că $\sum_{k=1}^{n} k = O(n)$. Desigur, $\sum_{k=1}^{1} k = O(1)$. Acceptând delimitarea pentru n, o demonstrăm acum pentru $n + 1$:

$$\begin{aligned} \sum_{k=1}^{n+1} k &= \sum_{k=1}^{n} k + (n+1) \\ &= O(n) + (n+1) \qquad \Longleftarrow \textit{greşit!} \\ &= O(n+1). \end{aligned}$$

Greşeala în argumentaţie este aceea că O ascunde o "constantă" care creşte în funcţie de n şi deci nu este constantă. Nu am arătat că aceeaşi constantă e valabilă pentru *toţi* n.

Delimitarea termenilor

Uneori, o limită superioară bună a unei serii se poate obţine delimitând fiecare termen al seriei, şi este, de multe ori, suficient să utilizăm cel mai mare termen pentru a-i delimita pe ceilalţi. De exemplu, o limită superioară rapidă a seriei aritmetice (3.1) este

$$\sum_{k=1}^{n} k \leq \sum_{k=1}^{n} n = n^2.$$

În general, pentru o serie $\sum_{k=1}^{n} a_k$, dacă punem $a_{max} = \max_{1 \leq k \leq n} a_k$, atunci

$$\sum_{k=1}^{n} a_k \leq n a_{max}.$$

Tehnica delimitării fiecărui termen dintr-o serie prin cel mai mare termen este o metodă slabă când seria poate fi, în fapt, delimitată printr-o serie geometrică. Fiind dată seria $\sum_{k=0}^{n} a_k$, să presupunem că $a_{k+1}/a_k \leq r$ pentru orice $k \geq 0$, unde $r < 1$ este o constantă. Suma poate fi delimitată printr-o serie geometrică descrescătoare infinită, deoarece $a_k \leq a_0 r^k$, şi astfel,

$$\sum_{k=0}^{n} a_k \leq \sum_{k=0}^{\infty} a_0 r^k = a_0 \sum_{k=0}^{\infty} r^k = a_0 \frac{1}{1-r}.$$

Putem aplica această metodă pentru a delimita suma $\sum_{k=1}^{\infty}(k/3^k)$. Primul termen este 1/3, iar raportul termenilor consecutivi este

$$\frac{(k+1)/3^{k+1}}{k/3^k} = \frac{1}{3}\cdot\frac{k+1}{k} \le \frac{2}{3},$$

pentru toţi $k \ge 1$. Astfel, fiecare termen este mărginit superior de $(1/3)(2/3)^k$, astfel încât

$$\sum_{k=1}^{\infty}\frac{k}{3^k} \le \sum_{k=1}^{\infty}\frac{1}{3}\left(\frac{2}{3}\right)^k = \frac{1}{3}\cdot\frac{1}{1-2/3} = 1.$$

O greşeală banală în aplicarea acestei metode este să arătăm că raportul termenilor consecutivi este mai mic decât 1 şi apoi să presupunem că suma este mărginită de o serie geometrică. Un exemplu este seria armonică infinită, care diverge, deoarece

$$\sum_{k=1}^{\infty}\frac{1}{k} = \lim_{n\to\infty}\sum_{k=1}^{n}\frac{1}{k} = \lim_{n\to\infty}\Theta(\lg n) = \infty.$$

Raportul termenilor $k+1$ şi k în această serie este $k/(k+1) < 1$, dar seria nu e mărginită de o serie geometrică descrescătoare. Pentru a delimita o serie printr-o serie geometrică, trebuie să arătăm că raportul este mărginit de o constantă subunitară; adică trebuie să existe un $r < 1$, care este *constant*, astfel încât raportul tuturor perechilor de termeni consecutivi nu depăşeşte niciodată r. În seria armonică nu există nici un astfel de r, deoarece raportul devine arbitrar de apropiat de 1.

Partiţionarea sumelor

Delimitări pentru o sumare dificilă se pot obţine exprimând seria ca o sumă de două sau mai multe serii, partiţionând domeniul indicelui şi apoi delimitând fiecare dintre seriile rezultate. De exemplu, să presupunem că încercăm să găsim o limită inferioară pentru seria aritmetică $\sum_{k=1}^{n} k$, despre care s-a arătat deja că are limita superioară n^2. Am putea încerca să minorăm fiecare termen al sumei prin cel mai mic termen, dar deoarece acest termen este 1, găsim o limită inferioară de n pentru sumă – foarte departe de limita noastră superioară, n^2.

Putem obţine o limită inferioară mai bună, partiţionând întâi suma. Să presupunem, pentru comoditate, că n este par. Avem

$$\sum_{k=1}^{n} k = \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \ge \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2) \ge (n/2)^2 = \Omega(n^2)$$

care este o limită asimptotic strânsă, deoarece $\sum_{k=1}^{n} k = O(n^2)$.

Pentru o sumă care apare din analiza unui algoritm, putem partiţiona adesea suma şi ignora un număr constant de termeni iniţiali. În general, această tehnică se aplică atunci când fiecare termen a_k din suma $\sum_{k=0}^{n} a_k$ este independent de n. Atunci pentru orice $k_0 > 0$ constant putem scrie

$$\sum_{k=0}^{n} a_k = \sum_{k=0}^{k_0-1} a_k + \sum_{k=k_0}^{n} a_k = \Theta(1) + \sum_{k=k_0}^{n} a_k,$$

deoarece termenii iniţiali ai sumei sunt toţi constanţi şi numărul lor este constant. Apoi, putem utiliza alte metode pentru a delimita $\sum_{k=k_0}^{n} a_k$. De exemplu, pentru a găsi o limită superioară asimptotică pentru

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k},$$

observăm că raportul termenilor consecutivi este

$$\frac{(k+1)^2/2^{k+1}}{k^2/2^k} = \frac{(k+1)^2}{2k^2} \leq \frac{8}{9},$$

dacă $k \geq 3$. Astfel, suma poate fi partiţionată în

$$\sum_{k=0}^{\infty} \frac{k^2}{2^k} = \sum_{k=0}^{2} \frac{k^2}{2^k} + \sum_{k=3}^{\infty} \frac{k^2}{2^k} \leq O(1) + \frac{9}{8} \sum_{k=0}^{\infty} \left(\frac{8}{9}\right)^k = O(1),$$

deoarece a doua sumă este o serie geometrică descrescătoare.

Tehnica partiţionării sumelor poate fi utilizată pentru a găsi delimitări asimptotice în situaţii mult mai dificile. De exemplu, putem obţine o delimitare $O(\lg n)$ a seriei armonice (3.5)

$$H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

Ideea este să partiţionăm domeniul de la 1 la n în $\lfloor \lg n \rfloor$ părţi şi să delimităm superior contribuţia fiecărei părţi prin 1. Astfel,

$$\sum_{k=1}^{n} \frac{1}{k} \leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} \leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \leq \lg n + 1. \tag{3.8}$$

Aproximarea prin integrale

Când o sumă poate fi exprimată ca $\sum_{k=m}^{n} f(k)$, unde $f(k)$ este o funcţie monoton crescătoare, o putem aproxima prin integrale:

$$\int_{m-1}^{n} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx. \tag{3.9}$$

Justificarea pentru această aproximare este arătată în figura 3.1. Suma este reprezentată prin aria dreptunghiurilor din figură, iar integrala este regiunea haşurată de sub curbă. Când $f(k)$ este o funcţie monoton descrescătoare, putem utiliza o metodă similară pentru a obţine marginile

$$\int_{m}^{n+1} f(x)dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)dx \tag{3.10}$$

Aproximaţia integrală (3.10) dă o estimare strânsă pentru al n-lea număr armonic. Pentru o limită inferioară, obţinem

$$\sum_{k=1}^{n} \frac{1}{k} \geq \int_{1}^{n+1} \frac{dx}{x} = \ln(n+1). \tag{3.11}$$

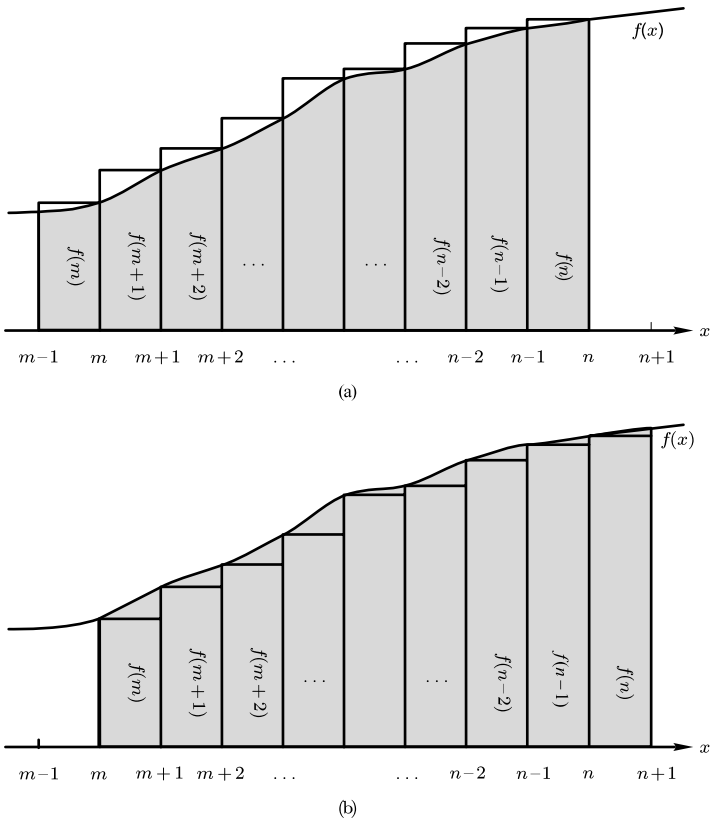


Figura 3.1 Aproximarea lui $\sum_{k=m}^{n} f(k)$ prin integrale. Aria fiecărui dreptunghi este indicată în interiorul dreptunghiului, iar aria tuturor dreptunghiurilor reprezintă valoarea sumei. Integrala este reprezentată prin aria haşurată de sub curbă. Comparând ariile în **(a)**, obţinem $\int_{m-1}^{n} f(x)dx \leq \sum_{k=m}^{n} f(k)$, şi apoi, deplasând dreptunghiurile cu o unitate la dreapta, obţinem $\sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x)dx$ din **(b)**.

Pentru limita superioară, deducem inegalitatea

$$\sum_{k=2}^{n} \frac{1}{k} \quad \leq \quad \int_{1}^{n} \frac{dx}{x} = \ln n,$$

care dă limita

$$\sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 \tag{3.12}$$

Exerciţii

3.2-1 Arătaţi că $\sum_{k=1}^{n} 1/k^2$ este mărginită superior de o constantă.

3.2-2 Găsiţi o margine superioară a sumei

$$\sum_{k=0}^{\lfloor \lg n \rfloor} \lceil n/2^k \rceil .$$

3.2-3 Arătaţi că al n-lea număr armonic este $\Omega(\lg n)$, partiţionând suma.

3.2-4 Aproximaţi $\sum_{k=1}^{n} k^3$ printr-o integrală.

3.2-5 De ce nu am utilizat aproximaţia integrală (3.10) direct pentru $\sum_{k=1}^{n} 1/k$ pentru a obţine o margine superioară pentru al n-lea număr armonic?

Probleme

3-1 Delimitarea sumelor
Daţi delimitări asimptotice strânse pentru următoarele sume. Presupuneţi că $r \geq 0$ şi $s \geq 0$ sunt constante.

a. $\displaystyle\sum_{k=1}^{n} k^r .$

b. $\displaystyle\sum_{k=1}^{n} \lg^s k .$

c. $\displaystyle\sum_{k=1}^{n} k^r \lg^s k .$

Note bibliografice

Knuth [121] este o referinţă excelentă pentru materialul prezentat în acest capitol. Proprietăţi de bază ale seriilor pot fi găsite în orice carte bună de analiză, cum ar fi Apostol [12] sau Thomas şi Finney [192].

4 Recurenţe

După cum s-a observat în capitolul 1, când un algoritm conţine o apelare recursivă la el însuşi, timpul său de execuţie poate fi descris adesea printr-o recurenţă. O ***recurenţă*** este o ecuaţie sau o inegalitate care descrie o funcţie exprimând valoarea sa pentru argumente mai mici. De exemplu, am văzut în capitolul 1 că timpul de execuţie în cazul cel mai defavorabil $T(n)$ al procedurii SORTEAZĂ-PRIN-INTERCLASARE poate fi descris prin recurenţa

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n = 1, \\ 2T(n/2) + \Theta(n) & \text{dacă } n > 1, \end{cases} \tag{4.1}$$

a cărei soluţie s-a afirmat că este $T(n) = \Theta(n \lg n)$.

Acest capitol oferă trei metode pentru rezolvarea recurenţelor – adică pentru obţinerea unor delimitări asimptotice "Θ" sau "O" ale soluţiei. În ***metoda substituţiei***, intuim o margine şi apoi utilizăm inducţia matematică pentru a demonstra că presupunerea noastră a fost corectă. ***Metoda iteraţiei*** converteşte recurenţa într-o sumă şi apoi se bazează pe tehnicile de delimitare a sumelor pentru a rezolva recurenţa. ***Metoda master*** furnizează delimitări pentru recurenţe de forma

$$T(n) = aT(n/b) + f(n),$$

unde $a \geq 1$, $b > 1$, iar $f(n)$ este o funcţie dată; ea cere memorarea a trei cazuri, dar după ce faceţi asta, determinarea marginilor asimptotice pentru multe recurenţe simple este uşoară.

Amănunte tehnice

În practică, neglijăm anumite detalii tehnice când formulăm şi rezolvăm recurenţe. Un bun exemplu de detaliu care adesea este omis este presupunerea că funcţiile au argumente întregi. În mod normal, timpul de execuţie $T(n)$ al unui algoritm este definit numai când n este un întreg, deoarece pentru majoritatea algoritmilor, dimensiunea datelor de intrare este întotdeauna un întreg. De exemplu, recurenţa care descrie timpul de execuţie în cazul cel mai favorabil al procedurii SORTEAZĂ-PRIN-INTERCLASARE este, de fapt,

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{dacă } n > 1. \end{cases} \tag{4.2}$$

Condiţiile la limită reprezintă o altă clasă de detalii pe care, de obicei, le ignorăm. Deoarece timpul de execuţie al unui algoritm pentru date de intrare de dimensiune constantă este o constantă, recurenţele care apar din timpii de execuţie ai algoritmilor au, în general, $T(n) = \Theta(1)$ pentru n suficient de mic. În consecinţă, pentru comoditate, vom omite, în general, afirmaţiile despre condiţiile la limită ale recurenţelor şi presupunem că $T(n)$ este constant pentru n mic. De exemplu, în mod normal formulăm recurenţa (4.1) ca

$$T(n) = 2T(n/2) + \Theta(n), \tag{4.3}$$

fără să dăm în mod explicit valori mici pentru n. Motivul este că, deşi schimbarea valorii lui $T(1)$ schimbă soluţia recurenţei, soluţia nu se modifică, în mod obişnuit, mai mult decât printr-un factor constant, astfel că ordinul de creştere este neschimbat.

Când formulăm şi rezolvăm recurenţe, omitem, adesea părţile întregi inferioare şi superioare şi condiţiile la limită. Mergem înainte fără aceste detalii şi apoi stabilim dacă ele contează sau nu. De obicei nu contează, dar este important să ştim dacă ele contează sau nu. Experienţa ajută, ajută şi unele teoreme care afirmă că aceste detalii nu afectează marginile asimptotice ale multor recurenţe întâlnite în analiza algoritmilor (vezi teorema 4.1 şi problema 4-5). În acest capitol, totuşi, vom aborda câteva dintre aceste detalii pentru a arăta unele dintre punctele sensibile ale metodelor de rezolvare a recurenţelor.

4.1. Metoda substituţiei

Metoda substituţiei pentru rezolvarea recurenţelor presupune intuirea formei soluţiei şi apoi utilizarea inducţiei matematice pentru a găsi constantele şi a arăta cum funcţionează soluţia. Numele provine din substituirea răspunsului intuit pentru funcţii când ipoteza inducţiei se aplică pentru valori mai mici. Metoda este puternică dar ea poate fi aplicată, în mod evident, numai în cazurile în care este uşor de intuit forma soluţiei.

Metoda substituţiei poate fi utilizată pentru a stabili fie margini superioare, fie margini inferioare pentru o recurenţă. De exemplu, să determinăm o limită superioară pentru recurenţa

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \tag{4.4}$$

care este similară cu recurenţele (4.2) şi (4.3). Intuim că soluţia este $T(n) = O(n \lg n)$. Metoda noastră constă în a demonstra că $T(n) \leq cn \lg n$ pentru o alegere corespunzătoare a constantei $c > 0$. Începem prin a presupune că această delimitare este valabilă pentru $\lfloor n/2 \rfloor$, adică $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituind în recurenţă, se obţine

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) \lg(\lfloor n/2 \rfloor)) + n \leq cn \lg(n/2) + n = cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \leq cn \lg n, \end{aligned}$$

unde ultimul pas este valabil cât timp $c \geq 1$.

Inducţia matematică ne cere acum să arătăm că soluţia îndeplineşte condiţiile la limită. Adică trebuie să arătăm că putem alege constanta c suficient de mare astfel încât delimitarea $T(n) \leq cn \lg n$ să funcţioneze şi pentru condiţiile la limită. Această cerinţă conduce uneori la probleme. Să presupunem că $T(1) = 1$ este singura condiţie la limită a recurenţei. Atunci, din nefericire, nu putem alege c suficient de mare, deoarece $T(1) \leq c1 \lg 1 = 0$.

Această dificultate în demonstrarea unei ipoteze de inducţie pentru o condiţie la limită precisă poate fi depăşită cu uşurinţă. Profităm de faptul că notaţia asimptotică ne cere doar să demonstrăm că $T(n) > cn \lg n$ pentru $n \geq n_0$ unde n_0 este o constantă. Ideea este să înlăturăm din discuţie condiţia la limită dificilă $T(1) = 1$ în demonstraţia inductivă şi să includem $n = 2$ şi $n = 3$ ca parte a condiţiilor la limită pentru demonstraţie. Putem impune $T(2)$ şi $T(3)$ ca şi condiţii la limită pentru demonstraţia inductivă, deoarece pentru $n > 3$, recurenţa nu se aplică în mod direct lui $T(1)$. Din recurenţă, deducem $T(2) = 4$ şi $T(3) = 5$. Demonstraţia inductivă că $T(n) \leq cn \lg n$ pentru o anumită constantă $c \geq 1$ poate fi completată acum, alegând c suficient de mare pentru ca $T(2) \leq c2 \lg 2$ şi $T(3) \leq c3 \lg 3$. După cum se dovedeşte, orice alegere a lui $c \geq 2$ este suficientă. Pentru majoritatea recurenţelor pe care le vom examina, extensia condiţiilor la limită pentru a face ipoteza de inducţie să funcţioneze pentru valori mici ale lui n este imediată.

Realizarea unei aproximări bune

Din nefericire, nu există o metodă generală de intuire a soluţiilor corecte ale unei recurenţe. Intuirea unei soluţii necesită experienţă şi, ocazional, creativitate. Din fericire, totuşi, există o anumită euristică care poate ajuta.

Dacă o recurenţă este similară cu una pe care aţi văzut-o deja, intuirea unei soluţii similare este rezonabilă. Ca exemplu, să considerăm recurenţa

$$T(n) = 2T(\lfloor n/2 \rfloor) + 17) + n,$$

care pare adevărată datorită adăugării lui "17" în argumentul lui T în membrul drept. Intuitiv, totuşi, acest termen suplimentar nu poate afecta în mod substanţial soluţia recurenţei. Când n este mare, diferenţa dintre $T(\lfloor n/2 \rfloor)$ şi $T(\lfloor n/2 \rfloor + 17)$ nu e atât de mare: ambele îl taie pe n aproximativ în două. În consecinţă, intuim că $T(n) = O(n \lg n)$, relaţie a cărei corectitudine se poate verifica utilizând metoda substituţiei (vezi exerciţiul 4.1-5).

Altă cale de a face o intuire bună este să demonstrăm limite superioare şi inferioare largi ale recurenţei şi apoi să reducem incertitudinea. De exemplu, putem începe cu o limită inferioară $T(n) = \Omega(n)$ pentru recurenţa (4.4), deoarece avem termenul n în recurenţă şi putem demonstra o limită superioară iniţială $T(n) = O(n^2)$. Apoi, putem coborî gradual limita superioară şi ridica limita inferioară până când ajungem la soluţia corectă, asimptotic strânsă, $T(n) = \Theta(n \lg n)$.

Subtilităţi

Există ocazii când puteţi intui corect o delimitare asimptotică a soluţiei unei recurenţe, dar, cumva, matematica nu pare să funcţioneze în inducţie. De regulă, problema este că ipoteza de inducţie nu este suficient de tare pentru a demonstra delimitarea detaliată. Când ajungeţi într-un astfel de impas, revizuirea aproximării prin scoaterea unui termen de ordin mai mic permite, adesea, matematicii să funcţioneze.

Să considerăm recurenţa

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$$

Intuim că soluţia este $O(n)$ şi încercăm să arătăm că $T(n) \leq cn$ pentru o alegere corespunzătoare a constantei c. Substituind presupunerea noastră în recurenţă, obţinem

$$T(n) \leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 = cn + 1,$$

care nu implică $T(n) \leq cn$ pentru orice alegere a lui c. Este tentant să încercăm o soluţie mai mare, să zicem $T(n) = O(n^2)$, care poate fi făcută să funcţioneze dar, în fapt, aproximarea noastră că soluţia este $T(n) = O(n)$ este corectă. Pentru a demonstra aceasta, totuşi, trebuie să facem o ipoteză de inducţie mai tare.

La prima vedere, aproximarea noastră e aproape corectă: avem în plus doar constanta 1, un termen de ordin inferior. Totuşi, inducţia matematică nu funcţionează decât dacă demonstrăm forma exactă a ipotezei de inducţie. Depăşim această dificultate scăzând un termen de ordin inferior din aproximarea noastră precedentă. Noua noastră intuire este $T(n) \leq cn - b$, unde $b \geq 0$ este constant. Avem acum

$$T(n) \leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1 = cn - 2b + 1 \leq cn - b,$$

cât timp $b \geq 1$. Ca şi mai înainte, constanta c trebuie să fie aleasă suficient de mare, pentru a îndeplini condiţiile la limită.

Majoritatea oamenilor găsesc ideea scăderii unui termen de ordin inferior contraintuitivă. La urma urmei, dacă matematica nu funcţionează, n-ar trebui să mărim soluţia intuită? Cheia în înţelegerea acestui pas este să ne amintim că utilizăm inducţia matematică: putem demonstra o proprietate mai puternică pentru o valoare dată acceptând o proprietate adevărată pentru valori mai mici.

Evitarea capcanelor

Este uşor să greşim în utilizarea notaţiei asimptotice. De exemplu, în recurenţa (4.4) putem demonstra în mod greşit că $T(n) = O(n)$ presupunând că $T(n) \leq cn$ şi apoi argumentând că

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \leq cn + n \\ &= O(n), \qquad \Longleftarrow \textit{ greşit!!} \end{aligned}$$

deoarece c este o constantă. Eroarea este că nu am demonstrat forma exactă a ipotezei de inducţie, adică faptul că $T(n) \leq cn$.

Schimbarea variabilelor

Uneori, o mică manipulare algebrică poate să transforme o recurenţă necunoscută într-una similară cunoscută. De exemplu, să considerăm recurenţa

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n,$$

care pare dificilă. Putem simplifica această recurenţă, totuşi, printr-o schimbare de variabile. Pentru comoditate, nu ne vom face griji în ceea ce priveşte rotunjirea valorilor, astfel încât ele să fie întregi, cum e cazul cu $\sqrt{n}$. Redenumind $m = \lg n$, obţinem

$$T(2^m) = 2T(2^{m/2}) + m.$$

Putem redenumi acum $S(m) = T(2^n)$ pentru a obţine nouă recurenţă

$$S(m) = 2S(m/2) + m,$$

care e foarte asemănătoare cu recurenţa (4.4) şi are aceeaşi soluţie: $S(m) = O(m \lg n)$. Revenind de la $S(m)$ la $T(n)$, obţinem $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

Exerciţii

4.1-1 Arătaţi că soluţia lui $T(n) = T(\lceil n/2 \rceil) + 1$ este $O(\lg n)$.

4.1-2 Arătaţi că soluţia lui $T(n) = 2T(\lfloor n/2 \rfloor) + n$ este $\Omega(n \lg n)$. Concluzionaţi că soluţia este $\Theta(n \lg n)$.

4.1-3 Arătaţi că făcând o ipoteză de inducţie diferită, putem depăşi dificultatea cu condiţia la limită $T(1) = 1$ pentru recurenţa (4.4) fără a ajusta condiţiile la limită pentru demonstraţia inductivă.

4.1-4 Arătaţi că $\Theta(n \lg n)$ este soluţia recurenţei "exacte" (4.2) pentru procedura SORTEAZĂ-PRIN-INTERCLASARE.

4.1-5 Arătaţi că soluţia lui $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ este $O(n \lg n)$.

4.1-6 Rezolvaţi recurenţa $T(n) = 2T(\sqrt{n}) + 1$ făcând o schimbare de variabile, indiferent dacă valorile sunt întregi sau nu.

4.2. Metoda iteraţiei

Metoda iterării unei recurenţe nu ne cere să aproximăm o soluţie dar poate necesita mai multe noţiuni de algebră decât metoda substituţiei. Ideea este să dezvoltăm (iterăm) recurenţa şi să o exprimăm ca o sumă de termeni care depind numai de n şi de condiţiile iniţiale. Se pot utiliza apoi tehnici de evaluare a sumelor pentru a găsi delimitări ale soluţiei.

De exemplu, să considerăm recurenţa

$$T(n) = 3T(\lfloor n/4 \rfloor) + n.$$

O iterăm după cum urmează:

$$\begin{aligned} T(n) &= n + 3T(\lfloor n/4 \rfloor) = n + 3(\lfloor n/4 \rfloor + 3T(\lfloor n/16 \rfloor)) \\ &= n + 3(\lfloor n/4 \rfloor + 3(\lfloor n/16 \rfloor + 3T(\lfloor n/64 \rfloor))) \\ &= n + 3\lfloor n/4 \rfloor + 9\lfloor n/16 \rfloor + 27T(\lfloor n/64 \rfloor), \end{aligned}$$

unde $\lfloor \lfloor n/4 \rfloor / 4 \rfloor = \lfloor n/16 \rfloor$ şi $\lfloor \lfloor n/16 \rfloor / 4 \rfloor = \lfloor n/64 \rfloor$ rezultă din identitatea (2.4).

Cât de departe trebuie să iterăm recurenţa pentru a ajunge la condiţia limită? Cel de-al i-lea termen din serie este $3^i \lfloor n/4^i \rfloor$. Iteraţia ajunge la $n = 1$ când $\lfloor n/4^i \rfloor = 1$ sau, echivalent, când i depăşeşte $\log_4 n$.

Continuând iterarea până în acest punct şi utilizând delimitarea $\lfloor n/4^i \rfloor \leq n/4^i$, descoperim că suma conţine o serie geometrică descrescătoare:

$$\begin{aligned} T(n) &\leq n + 3n/4 + 9n/16 + 27n/64 + \ldots + 3^{\log_4 n}\Theta(1) \\ &\leq n \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta\left(n^{\log_4 3}\right) = 4n + o(n) = O(n). \end{aligned}$$

Aici, am utilizat identitatea (2.9) pentru a concluziona că $3^{\log_4 n} = n^{\log_4 3}$ şi am utilizat faptul că $\log_4 3 < 1$ pentru a deduce că $\Theta(n^{\log_4 3}) = o(n)$.

Metoda iteraţiei duce de obicei la multe noţiuni de algebră, iar asigurarea corectitudinii calculelor poate fi o problemă. Cheia este să ne focalizăm asupra a doi parametri: numărul de recurenţe necesare pentru a ajunge la condiţia la limită şi suma termenilor care apar la fiecare nivel al procesului de iteraţie. Uneori, în procesul de iterare a unei recurenţe, puteţi intui soluţia fără a face toate calculele. Atunci iterarea poate fi abandonată în favoarea metodei substituţiei, care de regulă necesită mai puţine noţiuni de algebră.

Când o recurenţă conţine funcţiile parte întreagă inferioară şi superioară, calculele pot deveni deosebit de complicate. Adesea este util să presupunem că recurenţa este definită numai pe puterile exacte ale unui număr. În exemplul nostru, dacă am fi presupus că $n = 4^k$ pentru un

anumit întreg k, funcţia parte întreagă inferioară ar fi putut fi omisă, în mod convenabil. Din nefericire, demonstrarea delimitării $T(n) = O(n)$ numai pentru puterile exacte ale lui 4 este, din punct de vedere tehnic, un abuz al notaţiei O. Definiţiile notaţiei asimptotice pretind ca delimitările să fie demonstrate pentru *toţi* întregii suficient de mari, nu doar pentru acei care sunt puteri ale lui 4. Vom vedea în secţiunea 4.3 că pentru un mare număr de recurenţe această dificultate tehnică poate fi depăşită. Problema 4-5 oferă, de asemenea, condiţii în care o analiză pentru puterile exacte ale unui întreg poate fi extinsă la toţi întregii.

Arbori de recurenţă

Un ***arbore de recurenţă*** este un mod convenabil de a vizualiza ceea ce se întâmplă când o recurenţă este iterată şi poate să ajute la organizarea aparatului algebric necesar pentru a rezolva recurenţa. Este în mod special util când recurenţa descrie un algoritm divide şi stăpâneşte. Figura 4.1 arată deducerea arborelui de recurenţă pentru

$$T(n) = 2T(n/2) + n^2.$$

Pentru comoditate, presupunem că n este o putere exactă a lui 2. Partea (a) a figurii arată $T(n)$, care în partea (b) a fost dezvoltat într-un arbore echivalent care reprezintă recurenţa. Termenul n^2 este rădăcina (costul la nivelul cel mai de sus al recurenţei), iar cei doi subarbori ai rădăcinii sunt cele două recurenţe mai mici $T(n/2)$. Partea (c) arată acest proces dus cu un pas mai departe, dezvoltând $T(n/2)$. Costul pentru fiecare dintre cele două subnoduri de la al doilea nivel al recurenţei este $(n/2)^2$. Continuăm dezvoltarea fiecărui nod din arbore, spărgându-l în părţile componente, aşa cum sunt determinate de recurenţă, până ajungem la o condiţie la limită. Partea (d) arată arborele obţinut.

Evaluăm acum recurenţa adunând valorile de la fiecare nivel al arborelui. Nivelul cel mai înalt are valoarea totală n^2, nivelul al doilea are valoarea $(n/2)^2 + (n/2)^2 = n^2/2$, al treilea nivel are valoarea $(n/4)^2 + (n/4)^2 + (n/4)^2 + (n/4)^2 = n^2/4$ şi aşa mai departe. Deoarece valorile descresc geometric, totalul este cu cel mult un factor constant mai mare decât termenul cel mai mare (primul) şi deci soluţia este $\Theta(n^2)$.

Cu un alt exemplu, mai complicat, figura 4.2 arată arborele de recurenţă pentru

$$T(n) = T(n/3) + T(2n/3) + n.$$

(Pentru simplitate, omitem iarăşi funcţiile parte întreagă inferioară şi superioară). Când adăugăm valorile de la nivelurile arborelui de recurenţă, obţinem o valoare a lui n pentru fiecare nivel. Cel mai lung drum de la rădăcină la o frunză este $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$. Deoarece $(2/3)^k n = 1$ când $k = \log_{3/2} n$, înălţimea arborelui este $\log_{3/2} n$. Astfel, soluţia recurenţei este cel mult $n \log_{3/2} n = O(n \lg n)$.

Exerciţii

4.2-1 Determinaţi o limită superioară asimptotică bună pentru recurenţa $T(n) = 3T(\lfloor n/2 \rfloor) + n$ prin iteraţie.

4.2-2 Argumentaţi că soluţia recurenţei $T(n) = T(n/3) + T(2n/3) + n$ este $\Omega(n \lg n)$ făcând apel la un arbore de recurenţă.

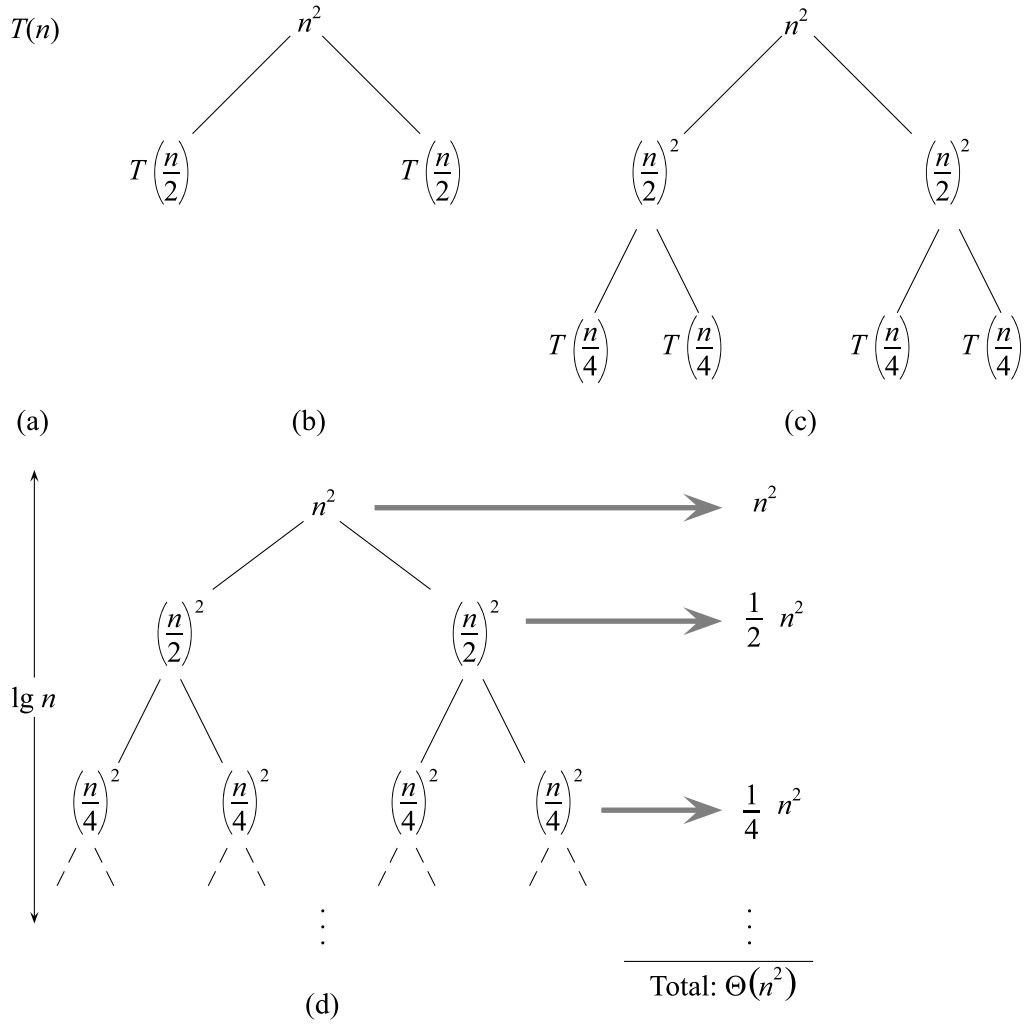


Figura 4.1 Construirea arborelui de recurenţă pentru $T(n) = 2T(n/2) + n^2$. Partea **(a)** îl prezintă pe $T(n)$ care în **(b)-(d)** este expandat treptat în trei arbori de recurenţă. Arborele expandat complet din partea (d) are înălţimea lg n (are lg $n + 1$ niveluri).

4.2-3 Determinaţi arborele de recurenţă pentru $T(n) = 4T(\lfloor n/2 \rfloor) + n$ şi daţi margini asimptotice strânse pentru soluţia sa.

4.2-4 Utilizaţi iteraţia pentru a rezolva recurenţa $T(n) = T(n-a) + T(a) + n$, unde $a \geq 1$ este o constantă.

4.2-5 Utilizaţi un arbore de recurenţă pentru a rezolva recurenţa $T(n) = T(\alpha n) + T((1-\alpha)n) + n$, unde α este o constantă din domeniul $0 < \alpha < 1$.

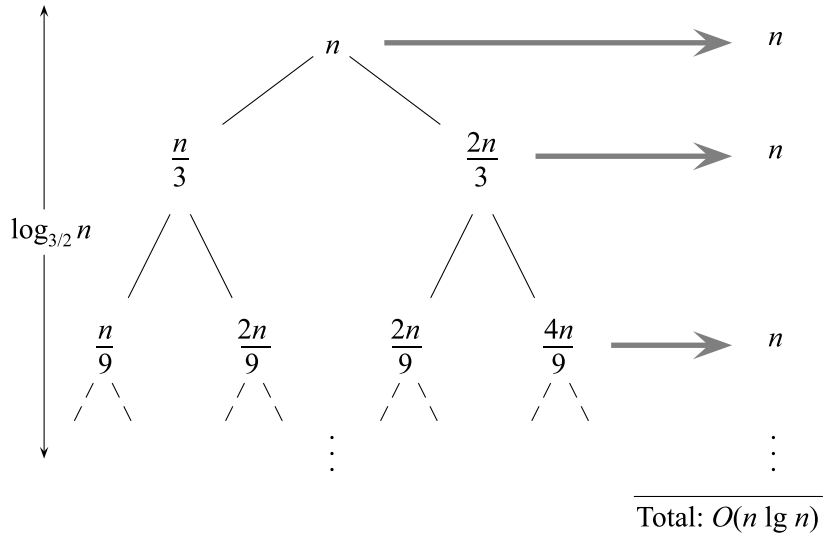


Figura 4.2 Un arbore de recurenţă pentru $T(n) = T(n/3) + T(2n/3) + n$.

4.3. Metoda master

Metoda master furnizează o "reţetă" pentru rezolvarea recurenţelor de forma

$$T(n) = aT(n/b) + f(n) \tag{4.5}$$

unde $a \geq 1$ şi $b > 1$ sunt constante, iar $f(n)$ este o funcţie asimptotic pozitivă. Metoda master pretinde memorarea a trei cazuri, dar apoi soluţia multor recurenţe se poate determina destul de uşor, de multe ori fără creion şi hârtie.

Recurenţa (4.5) descrie timpul de execuţie al unui algoritm care împarte o problemă de dimensiune n în a subprobleme, fiecare de dimensiune n/b, unde a şi b sunt constante pozitive. Cele a subprobleme sunt rezolvate recursiv, fiecare în timp $T(n/b)$. Costul divizării problemei şi al combinării rezultatelor subproblemelor este descris de funcţia $f(n)$. (Adică, utilizând notaţia din secţiunea 1.3.2, $f(n) = D(n) + C(n)$.) De exemplu, în recurenţa din procedura SORTEAZĂ-PRIN-INTERCLASARE $a = 2$, $b = 2$ şi $f(n) = \Theta(n)$.

Din punctul de vedere al corectitudinii tehnice, recurenţa nu este, de fapt, bine definită, deoarece n/b ar putea să nu fie un întreg. Înlocuirea fiecăruia dintre cei a termeni $T(n/b)$ fie cu $T(\lfloor n/b \rfloor)$ fie cu $T(\lceil n/b \rceil)$ nu afectează, totuşi, comportamentul asimptotic al recurenţei. (Vom demonstra aceasta în secţiunea următoare.) În mod normal ne va conveni, de aceea, să omitem funcţiile parte întreagă inferioară şi superioară când scriem recurenţe divide şi stăpâneşte de această formă.

Teorema master

Metoda master depinde de următoarea teoremă.

Teorema 4.1 (Teorema master) Fie $a \geq 1$ şi $b > 1$ constante, fie $f(n)$ o funcţie şi fie $T(n)$ definită pe întregii nenegativi prin recurenţa

$$T(n) = aT(n/b) + f(n)$$

unde interpretăm n/b fie ca $\lfloor n/b \rfloor$, fie ca $\lceil n/b \rceil$. Atunci $T(n)$ poate fi delimitată asimptotic după cum urmează.

1. Dacă $f(n) = O(n^{\log_b a - \epsilon})$ pentru o anumită constantă $\epsilon > 0$, atunci $T(n) = \Theta(n^{\log_b a})$.
2. Dacă $f(n) = \Theta(n^{\log_b a})$, atunci $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Dacă $f(n) = \Omega(n^{\log_b a + \epsilon})$ pentru o anumită constantă $\epsilon > 0$, şi dacă $af(n/b) \leq cf(n)$ pentru o anumită constantă $c < 1$ şi toţi n suficient de mari, atunci $T(n) = \Theta(f(n))$. ∎

Înainte de a aplica teorema master la câteva exemple, să ne oprim puţin ca să înţelegem ce spune. În fiecare dintre cele trei cazuri, comparăm funcţia $f(n)$ cu funcţia $n^{\log_b a}$. Intuitiv, soluţia recurenţei este determinată de cea mai mare dintre cele două funcţii. Dacă funcţia $n^{\log_b a}$ este mai mare, ca în cazul 1, atunci soluţia este $T(n) = \Theta(n^{\log_b a})$. Dacă funcţia $f(n)$ este mai mare, ca în cazul 3, atunci soluţia este $T(n) = \Theta(f(n))$. Dacă cele două funcţii sunt de acelaşi ordin de mărime, ca în cazul 2, înmulţim cu un factor logaritmic, iar soluţia este $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Dincolo de această intuiţie, există nişte detalii tehnice care trebuie înţelese. În primul caz, $f(n)$ trebuie nu numai să fie mai mică decât $n^{\log_b a}$, trebuie să fie *polinomial* mai mică. Adică $f(n)$ trebuie să fie asimptotic mai mică decât $n^{\log_b a}$ cu un factor n^{ϵ} pentru o anumită constantă $\epsilon > 0$. În al treilea caz, $f(n)$ trebuie nu numai să fie mai mare decât $n^{\log_b a}$, trebuie să fie polinomial mai mare şi, în plus, să verifice condiţia de "regularitate" $af(n/b) \leq cf(n)$. Această condiţie este satisfăcută de majoritatea funcţiilor polinomial mărginite pe care le vom întâlni.

Este important de realizat că cele trei cazuri nu acoperă toate posibilităţile pentru $f(n)$. Există un gol între cazurile 1 şi 2 când $f(n)$ este mai mic decât $n^{\log_b a}$, dar nu polinomial mai mic. Analog, există un gol între cazurile 2 şi 3, când $f(n)$ este mai mare decât $b^{\log_b a}$ dar nu polinomial mai mare. Dacă funcţia $f(n)$ cade într-unul dintre aceste goluri, sau când a condiţia de regularitate din cazul 3 nu este verificată, metoda master nu poate fi utilizată pentru a rezolva recurenţa.

Utilizarea metodei master

Pentru a utiliza metoda master, stabilim pur şi simplu care caz al teoremei se aplică (dacă e vreunul) şi scriem soluţia.

Ca un prim exemplu, să considerăm

$$T(n) = 9T(n/3) + n.$$

Pentru această recurenţă, avem $a = 9$, $b = 3$, $f(n) = n$ şi astfel $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Deoarece $f(n) = O(n^{\log_3 9 - \epsilon})$, unde $\epsilon = 1$, putem să aplicăm cazul 1 al teoremei master şi să considerăm că soluţia este $T(n) = \Theta(n^2)$.

Să considerăm acum

$$T(n) = T(2n/3) + 1,$$

în care $a = 1$, $b = 3/2$, $f(n) = 1$ şi $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Cazul 2 este cel care se aplică, deoarece $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$ şi astfel soluţia recurenţei este $T(n) = \Theta(\lg n)$.

Pentru recurenţa

$$T(n) = 3T(n/4) + n \lg n,$$

avem $a = 3$, $b = 4$, $f(n) = n \lg n$ şi $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Deoarece $f(n) = \Omega(n^{\log_4 3+\epsilon})$, unde $\epsilon \approx 0.2$, cazul 3 se aplică dacă putem arăta că pentru $f(n)$ este verificată condiţia de regularitate. Pentru n suficient de mare, $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ pentru $c = 3/4$. În consecinţă, din cazul 3, soluţia recurenţei este $T(n) = \Theta(n \lg n)$.

Metoda master nu se aplică recurenţei

$$T(n) = 2T(n/2) + n \lg n,$$

chiar dacă are forma potrivită: $a = 2$, $b = 2$, $f(n) = n \lg n$ şi $n^{\lg_b a} = n$. Se pare că ar trebui să se aplice cazul 3, deoarece $f(n) = n \lg n$ este asimptotic mai mare decât $n^{\log_b a} = n$, dar nu *polinomial* mai mare. Raportul $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ este asimptotic mai mic decât n^ϵ pentru orice constantă pozitivă ϵ. În consecinţă, recurenţa cade în golul dintre cazurile 2 şi 3. (Soluţia este dată în exerciţiul 4.4-2).

Exerciţii

4.3-1 Utilizaţi metoda master pentru a da o delimitare asimptotică strânsă pentru următoarele recurenţe.

a. $T(n) = 4T(n/2) + n$.

b. $T(n) = 4T(n/2) + n^2$.

c. $T(n) = 4T(n/2) + n^3$.

4.3-2 Timpul de execuţie al unui algoritm A este descris prin recurenţa $T(n) = 7T(n/2) + n^2$. Un algoritm concurent A' are timpul de execuţie de $T' = aT'(n/4) + n^2$. Care este cea mai mare valoare întreagă a lui a astfel încât A' este asimptotic mai rapid decât A?

4.3-3 Utilizaţi metoda master pentru a arăta că soluţia recurenţei $T(n) = T(n/2) + \Theta(1)$ a căutării binare (vezi exerciţiul 1.3-5) este $T(n) = \Theta(\lg n)$.

4.3-4 $\star$ Considerăm condiţia de regularitate $af(n/b) \leq cf(n)$ pentru o anumită constantă $c < 1$, care face parte din cazul 3 al teoremei master. Daţi un exemplu de o funcţie simplă $f(n)$ care satisface toate condiţiile din cazul 3 al teoremei master, cu excepţia condiţiei de regularitate.

4.4. Demonstraţia teoremei master

Această secţiune conţine o demonstraţie a teoremei master (teorema 4.1) pentru cititorii având cunoştinţe mai avansate. Pentru a aplica teorema nu este nevoie de înţelegerea demonstraţiei.

Demonstraţia constă din două părţi. Prima parte analizează recurenţa "master" (4.5) în ipoteza simplificatoare că $T(n)$ este definită numai pe puterile exacte ale lui $b > 1$, adică pentru $n = 1, b, b^2, \ldots$ Această parte furnizează toată intuiţia necesară pentru a înţelege de ce teorema master este adevărată. A doua parte arată de ce analiza se poate extinde la toţi întregii pozitivi n şi este, în esenţă, tehnică matematică aplicată la problema tratării părţilor întregi inferioare şi superioare.

În această secţiune, uneori vom abuza uşor de notaţia noastră asimptotică, utilizând-o pentru a descrie funcţii care sunt definite numai pe puteri exacte ale lui b. Reamintim că definiţiile notaţiilor asimptotice pretind ca delimitările să fie demonstrate pentru toate numerele suficient de mari, nu numai pentru cele care sunt puteri ale lui b. Deoarece am putea introduce noi notaţii asimptotice care să se aplice mulţimii $\{b^i : i = 0, 1, \ldots\}$ în locul întregilor nenegativi, acest abuz este minor.

Oricum, trebuie să fim întotdeauna puşi în gardă când utilizăm notaţii asimptotice pe un domeniu limitat, ca să nu tragem concluzii improprii. De exemplu, demonstrarea faptului că $T(n) = O(n)$ când n este o putere exactă a lui 2 nu garantează că $T(n) = O(n)$. Funcţia $T(n)$ poate fi definită ca

$$T(n) = \begin{cases} n & \text{dacă } n = 1, 2, 4, 8, \ldots, \\ n^2 & \text{în rest,} \end{cases}$$

caz în care cea mai bună margine superioară ce poate fi demonstrată este $T(n) = O(n^2)$. Din cauza acestui gen de consecinţe drastice, niciodată nu vom utiliza notaţia asimptotică pe un domeniu limitat fără să atragem atenţia asupra acestui lucru.

4.4.1. Demonstraţia pentru puteri exacte

Prima parte a demonstraţiei teoremei master analizează recurenţa master (4.5),

$$T(n) = aT(n/b) + f(n),$$

în ipoteza că n este o putere exactă a lui $b > 1$, unde n nu trebuie să fie un întreg. Analiza e împărţită în trei leme. Prima reduce problema rezolvării recurenţei master la problema evaluării unei expresii care conţine o sumă. A doua determină margini ale acestei sume. A treia lemă le pune împreună pe primele două pentru a demonstra o versiune a teoremei master pentru cazul în care n este o putere exactă a lui b.

Lema 4.2 Fie $a \geq 1$ şi $b > 1$ constante şi fie $f(n)$ o funcţie nenegativă definită pe puterile exacte ale lui b. Definim $T(n)$ pe puterile exacte ale lui b prin recurenţa

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n = 1, \\ aT(n/b) + f(n) & \text{dacă } n = b^i, \end{cases}$$

unde i este un întreg pozitiv. Atunci

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j). \tag{4.6}$$

Demonstraţie. Iterarea recurenţei conduce la

$$\begin{aligned} T(n) &= f(n) + aT(n/b) = f(n) + af(n/b) + a^2T(n/b^2) \\ &= f(n) + af(n/b) + a^2f(n/b^2) + \ldots + a^{\log_b n-1}f(n/b^{\log_b n-1}) + a^{\log_b n}T(1). \end{aligned}$$

Deoarece $a^{\log_b n} = n^{\log_b a}$, ultimul termen al acestei expresii devine

$$a^{\log_b n}T(1) = \Theta(n^{\log_b a}),$$

utilizând condiţia la limită $T(1) = \Theta(1)$. Termenii rămaşi pot fi exprimaţi sub forma sumei

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j);$$

astfel,

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f(n/b^j),$$

ceea ce completează demonstraţia. ∎

Arborele de recurenţă

Înainte de a continua, vom încerca să intuim puţin demonstraţia utilizând un arbore de recurenţă. Figura 4.3 arată arborele corespunzător iterării recurenţei din lema 4.2. Rădăcina arborelui are costul $f(n)$ şi are a fii, fiecare cu costul $f(n/b)$. (Este convenabil să ne gândim la a ca fiind un întreg, în special când vizualizăm arborele de recurenţă, dar din punct de vedere matematic nu este necesar acest lucru.) Fiecare dintre fii are a fii cu costul $f(n/b^2)$ şi astfel există a^2 noduri situate la distanţa 2 de rădăcină. În general, există a^j noduri care sunt la distanţa j de rădăcină şi fiecare are costul $f(n/b^j)$. Costul fiecărei frunze este $T(1) = \Theta(1)$ şi fiecare frunză este la distanţa $\log_b n$ de rădăcină, deoarece $n/b^{\log_b n} = 1$. Există $n^{\log_b n} = n^{\log_b a}$ frunze în arbore.

Putem obţine ecuaţia (4.6) însumând costurile fiecărui nivel al arborelui, după cum se arată în figură. Costul pentru un nivel j de noduri interne este $a^j f(n/b^j)$ şi astfel totalul pe toate nivelurile de noduri interne este

$$\sum_{j=0}^{\log_b n-1} a^j f(n/b^j).$$

În algoritmul divide şi stăpâneşte prezentat, această sumă reprezintă costurile divizării problemelor în subprobleme şi apoi al recombinării subproblemelor. Costul tuturor frunzelor, care este costul rezolvării tuturor celor $n^{\log_b a}$ subprobleme de dimensiune 1, este $\Theta(n^{\log_b a})$.

În limbajul arborelui de recurenţă, cele trei cazuri ale teoremei master corespund cazurilor în care costul total al arborelui este (1) dominat de costul frunzelor, (2) egal distribuit pe nivelurile arborelui sau (3) dominat de costul rădăcinii.

Suma din ecuaţia (4.6) descrie costul paşilor de divizare şi combinare din algoritmul divide şi stăpâneşte prezentat. Următoarea lemă furnizează delimitări asimptotice asupra creşterii sumei.

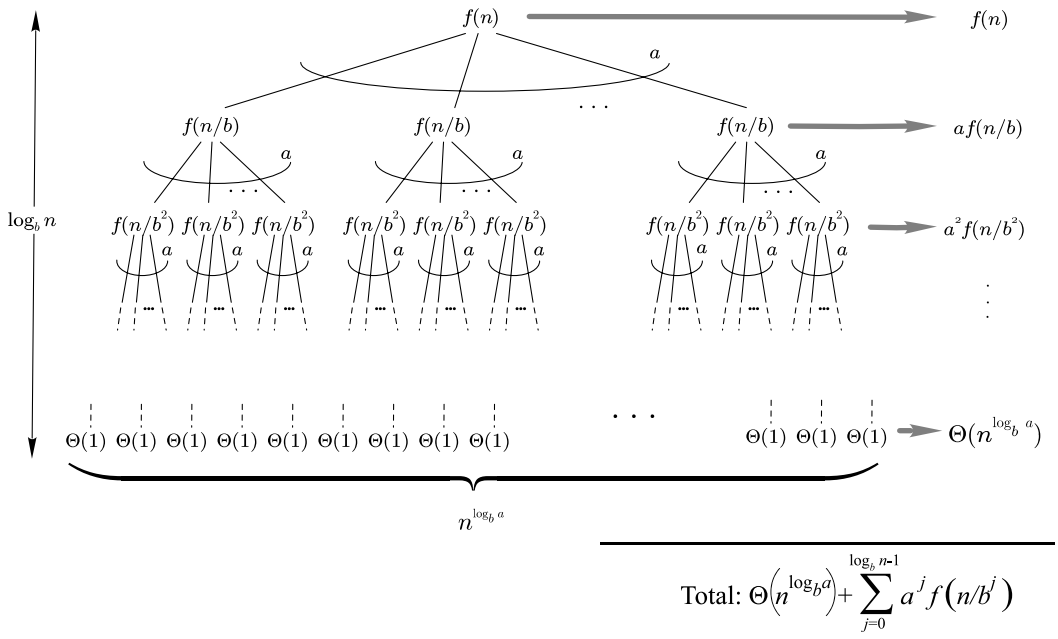


Figura 4.3 Arborele de recurenţă generat de $T(n) = aT(n/b) + f(n)$. Acesta este un arbore complet a-ar cu $n^{\log_b a}$ frunze, având înălţimea $\log_b n$. Costul fiecărui nivel este notat în dreapta, iar suma acestora este dată de ecuaţia (4.6)

Lema 4.3 Fie $a \geq 1$ şi $b > 1$ constante şi fie $f(n)$ o funcţie nenegativă definită pe puterile exacte ale lui b. O funcţie definită pe puterile exacte ale lui b prin

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \tag{4.7}$$

poate fi delimitată asimptotic pentru puterile exacte ale lui b după cum urmează.

1. Dacă $f(n) = O(n^{\log_b a - \epsilon})$ pentru o constantă $\epsilon > 0$, atunci $g(n) = O(n^{\log_b a})$.
2. Dacă $f(n) = \Theta(n^{\log_b a})$, atunci $g(n) = \Theta(n^{\log_b a} \lg n)$.
3. Dacă $af(n/b) \leq cf(n)$ pentru o anumită constantă $c < 1$ şi toţi $n \geq b$, atunci $g(n) = \Theta(f(n))$.

Demonstraţie. Pentru cazul 1, avem $f(n) = O(n^{\log_b a - \epsilon})$, de unde $f(n/b^j) = O((n/b^j)^{\log_b a - \epsilon})$. Înlocuind în ecuaţia (4.7) rezultă

$$g(n) = O\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j}\right)^{\log_b a - \epsilon}\right). \tag{4.8}$$

Delimităm suma din interiorul O-notaţiei factorizând termenii şi simplificând, rezultând o serie geometrică crescătoare

$$\begin{aligned}\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a-\epsilon} &= n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} \left(\frac{ab^\epsilon}{b^{\log_b a}}\right)^j = n^{\log_b a-\epsilon} \sum_{j=0}^{\log_b n-1} (b^\epsilon)^j \\ &= n^{\log_b a-\epsilon} \left(\frac{b^{\epsilon \log_b n}-1}{b^\epsilon-1}\right) = n^{\log_b a-\epsilon} \left(\frac{n^\epsilon-1}{b^\epsilon-1}\right).\end{aligned}$$

Deoarece b şi ϵ sunt constante, ultima expresie se reduce la $n^{\log_b a-\epsilon}O(n^\epsilon) = O(n^{\log_b a})$. Înlocuind această expresie în suma din ecuaţia (4.8), obţinem

$$g(n) = O\left(n^{\log_b a}\right),$$

şi cazul 1 este demonstrat.

În ipoteza că $f(n) = \Theta(n^{\log_b a})$ pentru cazul 2, avem $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Înlocuind în ecuaţia (4.7) rezultă

$$g(n) = \Theta\left(\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a}\right) \tag{4.9}$$

Delimităm suma din interiorul lui Θ ca şi în cazul 1, dar de data aceasta nu obţinem o serie geometrică. În schimb, descoperim că fiecare termen al sumei este acelaşi

$$\sum_{j=0}^{\log_b n-1} a^j \left(\frac{n}{b^j}\right)^{\log_b a} = n^{\log_b a} \sum_{j=0}^{\log_b n-1} \left(\frac{a}{b^{\log_b a}}\right)^j = n^{\log_b a} \sum_{j=0}^{\log_b n-1} 1 = n^{\log_b a} \log_b n.$$

Înlocuind această expresie pentru sumă în ecuaţia (4.9) rezultă

$$g(n) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \lg n),$$

şi cazul 2 este demonstrat.

Cazul 3 se demonstrează similar. Deoarece $f(n)$ apare în definiţia (4.7) a lui $g(n)$ şi toţi termenii lui $g(n)$ sunt nenegativi, putem concluziona că $g(n) = \Omega(f(n))$ pentru puterile exacte ale lui b. În ipoteza că $af(n/b) \le cf(n)$ pentru o anumită constantă $c < 1$ şi toţi $n \ge b$, avem $a^j f(n/b^j) \le c^j f(n)$.

Substituind în ecuaţia (4.7) şi simplificând, rezultă o serie geometrică, dar spre deosebire de seria din cazul 1, aceasta are termeni descrescători

$$g(n) \le \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \le \sum_{j=0}^{\log_b n-1} c^j f(n) \le f(n) \sum_{j=0}^{\infty} c^j = f(n) \frac{1}{1-c} = O(f(n)),$$

deoarece c este o constantă. Astfel, putem concluziona că $g(n) = \Theta(f(n))$ pentru puterile exacte ale lui b. Cazul 3 este demonstrat, ceea ce completează demonstraţia lemei. ∎

Putem demonstra acum o versiune a teoremei master pentru cazul în care n este o putere exactă a lui b.

Lema 4.4 Fie $a \geq 1$ şi $b > 1$ constante şi fie $f(n)$ o funcţie nenegativă, definită pe puterile exacte ale lui b. Definim $T(n)$ pe puterile exacte ale lui b prin recurenţa

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n = 1, \\ aT(n/b) + f(n) & \text{dacă } n = b^i, \end{cases}$$

unde i este un întreg pozitiv. Atunci $T(n)$ poate fi delimitat asimptotic pentru puterile exacte ale lui b după cum urmează

1. Dacă $f(n) = O(n^{\log_b a - \epsilon})$ pentru o anumită constantă $\epsilon > 0$, atunci $T(n) = \Theta(n^{\log_b a})$.
2. Dacă $f(n) = \Theta(n^{\log_b a})$, atunci $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Dacă $f(n) = \Omega(n^{\log_b a + \epsilon})$ pentru o anumită constantă $\epsilon > 0$ şi dacă $af(n/b) \leq cf(n)$ pentru o anumită constantă $c < 1$ şi toate valorile lui n suficient de mari, atunci $T(n) = \Theta(f(n))$.

Demonstraţie. Utilizăm delimitările din lema 4.3 pentru a evalua suma (4.6) din lema 4.2. Pentru cazul 1, avem

$$T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a}),$$

iar pentru cazul 2,

$$T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_b a} \lg n).$$

Pentru cazul 3, condiţia $af(n/b) \leq cf(n)$ implică $f(n) = \Omega(n^{\log_b a + \epsilon})$ (vezi exerciţiul 4.4-3). În consecinţă,

$$T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n)).$$

Cu aceasta lema este demonstrată. ∎

4.4.2. Părţi întregi inferioare şi superioare

Pentru a demonstra teorema master, trebuie să extindem acum analiza noastră la situaţia în care în recurenţa master sunt utilizate părţi întregi inferioare şi superioare, astfel că recurenţa este definită pentru orice numere întregi, nu doar pentru puterile exacte ale lui b. Obţinerea unei margini inferioare pentru

$$T(n) = aT(\lceil n/b \rceil) + f(n) \tag{4.10}$$

şi a unei margini superioare pentru

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \tag{4.11}$$

este o operaţie de rutină, deoarece delimitarea $\lceil n/b \rceil \geq nb$ poate fi utilizată în primul caz pentru a obţine rezultatul dorit, iar delimitarea $\lfloor n/b \rfloor \leq n/b$ poate fi utilizată în al doilea caz. Delimitarea inferioară a recurenţei (4.11) necesită în mare parte acelaşi procedeu ca şi delimitarea superioară a recurenţei (4.10), astfel că vom prezenta numai ultima delimitare.

Vrem să iterăm recurenţa (4.10) aşa cum s-a făcut în lema 4.2. Pe măsură ce iterăm recurenţa, obţinem un şir de invocări recursive cu argumentele

$$\begin{array}{l} n, \lceil n/b \rceil, \\ \lceil \lceil n/b \rceil / b \rceil, \\ \lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil, \\ \vdots \end{array}$$

Să notăm al i-lea element din şir cu n_i, unde

$$n_i = \begin{cases} n & \text{dacă } i = 0, \\ \lceil n_{i-1}/b \rceil & \text{dacă } i > 0. \end{cases} \tag{4.12}$$

Primul nostru scop este să determinăm numărul k de iteraţii astfel încât n_k să fie o constantă. Utilizând inegalitatea $\lceil x \rceil \leq x + 1$, obţinem

$$\begin{array}{l} n_0 \leq n, \\ n_1 \leq \frac{n}{b} + 1, \\ n_2 \leq \frac{n}{b^2} + \frac{1}{b} + 1, \\ n_3 \leq \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1, \\ \vdots \end{array}$$

În general,

$$n_i \leq \frac{n}{b^i} + \sum_{j=0}^{i-1} \frac{1}{b^j} \leq \frac{n}{b^i} + \frac{b}{b-1},$$

şi astfel, când $i = \lfloor log_b n \rfloor$, obţinem $n_i \leq b + b/(b-1) = O(1)$.

Putem itera acum recurenţa (4.10), obţinând

$$\begin{aligned} T(n) &= f(n_0) + aT(n_1) = f(n_0) + af(n_1) + a^2 T(n_2) \\ &\leq f(n_0) + af(n_1) + a^2 f(n_2) + \ldots + a^{\lfloor \log_b n \rfloor - 1} f(n_{\lfloor \log_b n \rfloor - 1}) + a^{\lfloor \log_b n \rfloor} T(n_{\lfloor \log_b n \rfloor}) \\ &= \Theta(n^{\log_b a}) + \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j g(n_j) \end{aligned} \tag{4.13}$$

care este foarte asemănătoare cu ecuaţia (4.6) cu excepţia faptului că n este un întreg arbitrar şi nu e restrâns la o putere exactă a lui b.

Putem evalua acum suma

$$g(n) = \sum_{j=0}^{\lfloor \log_b n \rfloor - 1} a^j f(n_j) \tag{4.14}$$

din (4.13) într-o manieră similară cu demonstraţiei lemei 4.3. Începând cu cazul 3, dacă $af(\lceil n/b \rceil) \leq cf(n)$ pentru $n > b + b/(b-1)$, unde $c < 1$ este o constantă, atunci rezultă că $a^j f(n_j) \leq c^j f(n)$. De aceea, suma din ecuaţia (4.14) poate fi evaluată exact ca în lema 4.3. Pentru cazul 2, avem $f(n) = \Theta(n^{\log_b a})$. Dacă putem arăta că $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$, atunci demonstraţia este valabilă şi în acest caz. Observăm că $j \leq \lfloor \log_b n \rfloor$ implică $b^j/n \leq 1$.

Delimitarea $f(n) = O(n^{\log_b a})$ implică existenţa unei constante $c > 0$ astfel încât pentru n_j suficient de mare,

$$\begin{aligned} f(n_j) &\leq c\left(\frac{n}{b^j} + \frac{b}{b-1}\right)^{\log_b a} = c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \left(\frac{b^j}{n} \cdot \frac{b}{b-1}\right)\right)^{\log_b a} \\ &= c\left(\frac{n^{\log_b a}}{a^j}\right)\left(1 + \frac{b}{b-1}\right)^{\log_b a} \leq O\left(\frac{n^{\log_b a}}{a^j}\right), \end{aligned}$$

deoarece $c(1 + b/(b-1))^{\log_b a}$ este o constantă. Astfel, cazul 2 este demonstrat. Demonstraţia cazului 1 este aproape identică. Cheia este să demonstrăm delimitarea $f(n_j) = O(n^{\log_b a - \epsilon})$, care este similară cu demonstraţia corespunzătoare din cazul 2, deşi calculele sunt mai complicate.

Am demonstrat delimitările superioare din teorema master pentru toţi întregii n. Demonstraţia delimitărilor inferioare este similară.

Exerciţii

4.4-1 ⋆ Daţi o expresie simplă şi exactă pentru n_i în ecuaţia (4.12) pentru cazul în care b este un întreg pozitiv în loc de un număr real arbitrar.

4.4-2 ⋆ Arătaţi că dacă $f(n) = \Theta(n^{\log_b a} \lg^k n)$, unde $k \geq 0$, atunci recurenţa master are soluţia $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Pentru simplitate, limitaţi-vă analiza la cazul puterilor exacte ale lui b.

4.4-3 Arătaţi că ipotezele cazului 3 din teorema master sunt abundente, în sensul că cerinţa de regularitate $af(n/b) \leq cf(n)$ pentru o anumită constantă $c < 1$ implică existenţa unei constante $\epsilon > 0$ astfel încât $f(n) = \Omega(n^{\log_b a + \epsilon})$.

Probleme

4-1 Exemple de recurenţe

Daţi margini asimptotice superioare şi inferioare pentru fiecare dintre următoarele recurenţe. Presupuneţi că $T(n)$ este constantă pentru $n \leq 2$. Faceţi marginile cât de strânse se poate şi justificaţi a răspunsurile.

a. $T(n) = 2T(n/2) + n^3$.

b. $T(n) = T(9n/10) + n$.

c. $T(n) = 16T(n/4) + n^2$.

d. $T(n) = 7T(n/3) + n^2$.

e. $T(n) = 7T(n/2) + n^2$.

f. $T(n) = 2T(n/4) + \sqrt{n}$.

g. $T(n) = T(n-1) + n$.

h. $T(n) = T(\sqrt{n}) + 1$.

4-2 Găsirea întregului lipsă
Un tablou $A[1..n]$ conţine toate valorile întregi de la 0 la n, cu excepţia unuia. Ar fi uşor să determinăm întregul lipsă în timp $O(n)$ utilizând un tablou auxiliar $B[0..n]$ pentru a înregistra numerele care apar în A. În această problemă, totuşi, accesul complet la un element din A nu se poate realiza printr-o singură operaţie. Elementele lui A sunt reprezentate în binar, iar singura operaţie pe care o putem utiliza pentru a le accesa este "extrage al j-lea bit al lui $A[i]$", care cere timp constant.

Arătaţi că dacă utilizăm numai această operaţie încă se mai poate determina întregul lipsă în timp $O(n)$.

4-3 Costul transmiterii parametrilor
Pe parcursul acestei cărţi presupunem că transmiterea parametrilor în timpul apelărilor procedurilor cere timp constant, chiar dacă este transmis un tablou cu N elemente. Această ipoteză este validă în majoritatea sistemelor, deoarece nu este transmis tabloul însuşi, ci un pointer către tablou. Această problemă examinează implicaţiile a trei strategii de transmitere a parametrilor:

1. Un tablou este transmis prin pointer. Timpul este $\Theta(1)$.
2. Un tablou este transmis prin copiere. Timpul este $\Theta(N)$, unde N este dimensiunea tabloului.
3. Un tablou este transmis copiind numai subdomeniul care ar putea fi accesat de procedura apelată. Timpul este $\Theta(p - q + 1)$ dacă este transmis subtabloul $A[p..q]$.

a. Consideraţi algoritmul recursiv de căutare binară pentru găsirea unui număr într-un tablou sortat (vezi exerciţiul 1.3-5). Daţi recurenţe pentru timpul de execuţie în cazul cel mai defavorabil când tablourile sunt transmise utilizând fiecare dintre cele trei metode de mai sus şi daţi margini superioare bune pentru soluţiile recurenţelor. Presupunem că dimensiunea problemei iniţiale este N, iar dimensiunea subproblemei este n.

b. Refaceţi partea (a) pentru algoritmul SORTEAZĂ-PRIN-INTERCLASARE din secţiunea 1.3.1.

4-4 Alte exemple de recurenţe
Daţi margini asimptotice superioare şi inferioare pentru $T(n)$ pentru fiecare dintre următoarele recurenţe. Presupuneţi că $T(n)$ este constantă pentru $n \leq 8$. Faceţi marginile cât de strânse se poate şi justificaţi răspunsurile.

a. $T(n) = 3T(n/2) + n \lg n$.

b. $T(n) = 3T(n/3 + 5) + n/2$.

c. $T(n) = 2T(n/2) + n/\lg n$.

d. $T(n) = T(n - 1) + 1/n$.

e. $T(n) = T(n - 1) + \lg n$.

f. $T(n) = \sqrt{n}T(\sqrt{n}) + n$.

4-5 Condiţii de pantă

Adesea suntem în stare să delimităm o recurenţă $T(n)$ pentru puterile exacte ale unei constante întregi b. Această problemă ne dă condiţii suficiente pentru a extinde delimitarea la toate valorile $n > 0$ reale.

a. Fie $T(n)$ şi $h(n)$ funcţii monoton crescătoare şi să presupunem că $T(n) \leq h(n)$ când n este o putere exactă a unei constante $b > 1$. Mai mult, să presupunem că $h(n)$ este "lent crescătoare", în sensul că $h(n) = O(h(n/b))$. Demonstraţi că $T(n) = O(h(n))$.

b. Să presupunem că avem recurenţa $T(n) = aT(n/b) + f(n)$, unde $a \geq 1$, $b > 1$, iar $f(n)$ este monoton crescătoare. Totodată condiţiile iniţiale pentru recurenţă sunt date de $T(n) = g(n)$ pentru $n \leq n_0$, unde $g(n)$ este monoton crescătoare, iar $g(n_0) \leq aT(n_0/b) + f(n_0)$. Demonstraţi că $T(n)$ este monoton crescătoare.

c. Simplificaţi demonstraţia teoremei master pentru cazul în care $f(n)$ este monoton crescătoare şi lent crescătoare. Utilizaţi lema 4.4.

4-6 Numere Fibonacci

Această problemă dezvoltă proprietăţi ale numerolor lui Fibonacci, care sunt definite prin recurenţa (2.13). Vom utiliza tehnica funcţiilor generatoare pentru a rezolva recurenţa lui Fibonacci. Să definim ***funcţia generatoare*** (sau ***seria formală de puteri***) $\mathcal{F}$ ca

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F_i z^i = 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \cdots .$$

a. Arătaţi că $\mathcal{F}(z) = z + \mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Arătaţi că

$$\mathcal{F}(z) = \frac{z}{1 - z - z^2} = \frac{z}{(1 - \phi z)(1 - \widehat{\phi} z)} = \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi z} - \frac{1}{1 - \widehat{\phi} z} \right),$$

unde

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.61803\ldots \text{ şi } \widehat{\phi} = \frac{1 - \sqrt{5}}{2} = -0.61803\ldots .$$

c. Arătaţi că

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \widehat{\phi}^i) z^i .$$

d. Demonstraţi că $F_i = \phi^i / \sqrt{5}$ pentru $i > 0$, rotunjit la cel mai apropiat întreg. (*Indicaţie:* $|\widehat{\phi}| < 1$.)

e. Demonstraţi că $F_{i+2} \geq \phi^i$ pentru $i \geq 0$.

4-7 Testarea chip-urilor VLSI

Profesorul Diogenes are n chip-uri VLSI[1] presupuse a fi identice, care sunt capabile în principiu să se testeze unele pe altele. Matriţa de testare a profesorului utilizează două chip-uri deodată. Când matriţa este încărcată, fiecare chip îl testează pe celălalt şi raportează dacă este bun sau defect. Un chip bun raportează întotdeauna corect dacă celălalt cip este bun sau defect, dar răspunsul unui chip defect nu este demn de încredere. Astfel, cele patru rezultate posibile ale testului sunt după cum urmează:

Chip-ul A spune	Chip-ul B spune	Concluzie
B este bun	A este bun	ambele sunt bune sau ambele sunt defecte
B este bun	A este defect	cel puţin unul este defect
B este defect	A este bun	cel puţin unul este defect
B este defect	A este defect	cel puţin unul este defect

a. Demonstraţi că dacă mai mult de de $n/2$ cip-uri sunt defecte, profesorul nu poate neapărat să determine care cip-uri sunt bune, utilizând orice strategie bazată pe acest tip de testare pe perechi. Presupuneţi că cip-urile defecte pot conspira pentru a-l păcăli pe profesor.

b. Să considerăm problema găsirii unui singur cip bun dintre n cip-uri, presupunând că mai mult de $n/2$ cip-uri sunt bune. Arătaţi că $\lfloor n/2 \rfloor$ teste pe perechi sunt suficiente pentru a reduce dimensiunea problemei la jumătate.

c. Arătaţi că cip-urile bune pot fi identificate cu $\Theta(n)$ teste pe perechi, presupunând că mai mult de $n/2$ cip-uri sunt bune. Daţi şi rezolvaţi recurenţa care descrie numărul de teste.

Note bibliografice

Recurenţele au fost studiate încă din 1202 de către L. Fibonacci, după care sunt denumite numerele lui Fibonacci. A. De Moivre a introdus metoda funcţiilor generatoare (vezi problema 4-6) pentru rezolvarea recurenţelor. Metoda master este adaptată din Bentley, Haken şi Saxe [26], care prezintă metoda extinsă justificată prin exerciţiul 4.4-2. Knuth [121] şi Liu [140] arată cum se rezolvă recurenţele liniare utilizând metoda funcţiilor generatoare. Purdom şi Brown [164] conţine o discuţie extinsă asupra rezolvării recurenţelor.

[1] VLSI înseamnă "very-large-scale-integration" (integrare la scară foarte mare), care este tehnologia chip-urilor de circuite integrate utilizată astăzi pentru fabricarea majorităţii microprocesoarelor.

5 Mulţimi etc.

În capitolele anterioare am studiat unele noţiuni matematice. În acest capitol vom recapitula şi vom completa notaţiile, definiţiile şi proprietăţile elementare ale mulţimilor, relaţiilor, funcţiilor, grafurilor şi arborilor. Cititorii familiarizaţi cu aceste noţiuni trebuie, doar, să răsfoiască paginile acestui capitol.

5.1. Mulţimi

O ***mulţime*** este o colecţie de obiecte distincte numite ***membri*** sau ***elemente***. Dacă un obiect x este element al mulţimii S, scriem $x \in S$ şi citim "x este un element al lui S" sau, mai scurt, "x aparţine lui S". Dacă x nu este un element al lui S, scriem $x \notin S$. Putem descrie o mulţime enumerându-i elementele în mod explicit ca o listă între acolade. De exemplu, putem defini o mulţime S care conţine numerele 1, 2 şi 3 scriind $S = \{1, 2, 3\}$. Deoarece 2 este un element al mulţimii S, vom scrie $2 \in S$, dar vom scrie $4 \notin S$ deoarece 4 nu este un element al lui S. O mulţime nu poate conţine acelaşi obiect de mai multe ori şi elementele sale nu sunt ordonate. Două mulţimi A şi B sunt ***egale*** dacă ele conţin aceleaşi elemente. Notaţia folosită pentru a arăta că două mulţimi sunt egale este $A = B$. De exemplu, avem $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

Folosim notaţii speciale pentru câteva mulţimi mai des întâlnite.

- Notăm cu $\emptyset$ ***mulţimea vidă***, adică mulţimea care nu conţine nici un element.
- Notăm cu $\mathbb{Z}$ mulţimea ***numerelor întregi***, adică mulţimea $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- Notăm cu $\mathbb{R}$ mulţimea ***numerelor reale***.
- Notăm cu $\mathbb{N}$ mulţimea ***numerelor naturale***, adică mulţimea $\{0, 1, 2, \ldots\}$.[1]

Dacă toate elementele unei mulţimi A sunt conţinute într-o mulţime B, adică dacă $x \in A$ implică $x \in B$, atunci scriem $A \subseteq B$ şi spunem că A este o ***submulţime*** a lui B. O mulţime A este o ***submulţime strictă*** a lui B, notat $A \subset B$, dacă $A \subseteq B$ dar $A \neq B$. (Unii autori folosesc simbolul "$\subset$" pentru a nota relaţia obişnuită de incluziune şi nu relaţia de incluziune strictă.) Pentru orice mulţime A, avem $A \subseteq A$. Pentru două mulţimi A şi B, avem $A = B$ dacă, şi numai dacă, $A \subseteq B$ şi $B \subseteq A$. Pentru oricare trei mulţimi A, B şi C, dacă $A \subseteq B$ şi $B \subseteq C$, atunci $A \subseteq C$. Pentru orice mulţime A avem $\emptyset \subseteq A$.

Uneori definim mulţimi pornind de la alte mulţimi deja definite. Dându-se o mulţime A, putem defini o mulţime $B \subseteq A$ impunând o restricţie care să distingă elementele din B de celelalte elemente din A. De exemplu putem defini mulţimea numerelor întregi pare prin

$$x : x \in \mathbb{Z}$$

şi $x/2$ este un număr întreg}. Cele două puncte din notaţia anterioară înseamnă "astfel încât." (Unii autori folosesc o bară verticală în locul acestor două puncte.)

[1] Unii autori susţin că 0 nu face parte din mulţimea numerelor naturale. Totuşi tendinţa modernă este de a include şi numărul 0 în această mulţime.

Dându-se două mulţimi A şi B putem defini noi mulţimi aplicând unii ***operatori cu mulţimi***:

- ***Intersecţia*** mulţimilor A şi B este mulţimea

$$A \cap B = \{x : x \in A \text{ şi } x \in B\}.$$

- ***Reuniunea*** mulţimilor A şi B este mulţimea

$$A \cup B = \{x : x \in A \text{ sau } x \in B\}.$$

- ***Diferenţa*** dintre două mulţimi A şi B este mulţimea

$$A - B = \{x : x \in A \text{ şi } x \notin B\}.$$

Operaţiile cu mulţimi respectă următoarele reguli:

Reguli ale mulţimii vide:

$$A \cap \emptyset = \emptyset,$$
$$A \cup \emptyset = A.$$

Reguli de idempotenţă:

$$A \cap A = A,$$
$$A \cup A = A.$$

Reguli de comutativitate:

$$A \cap B = B \cap A,$$
$$A \cup B = B \cup A.$$

Reguli de asociativitate:

$$A \cap (B \cap C) = (A \cap B) \cap C,$$
$$A \cup (B \cup C) = (A \cup B) \cup C.$$

Reguli de distributivitate:

$$\begin{aligned} A \cap (B \cup C) &= (A \cap B) \cup (A \cap C), \\ A \cup (B \cap C) &= (A \cup B) \cap (A \cup C). \end{aligned} \tag{5.1}$$

Reguli de absorbţie:

$$A \cap (A \cup B) = A,$$
$$A \cup (A \cap B) = A.$$

Legile lui DeMorgan:

$$\begin{aligned} A - (B \cap C) &= (A - B) \cup (A - C), \\ A - (B \cup C) &= (A - B) \cap (A - C). \end{aligned} \tag{5.2}$$

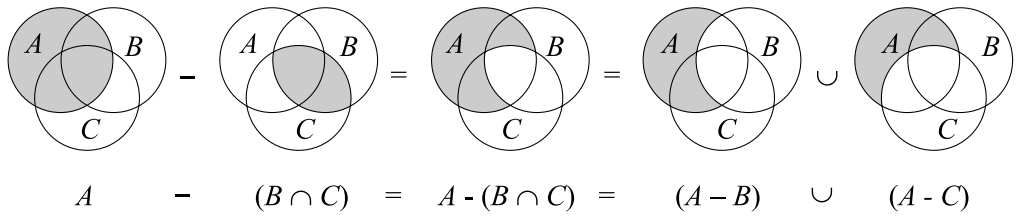


Figura 5.1 O diagramă Venn ilustrând prima regulă a lui DeMorgan (5.2). Fiecare dintre mulţimile A, B şi C este reprezentată ca un cerc în plan.

Prima dintre regulile lui DeMorgan este ilustrată în figura 5.1, folosind o ***diagramă Venn***, o imagine grafică în care mulţimile sunt reprezentate ca regiuni în plan.

Deseori toate mulţimile considerate sunt submulţimi ale unei mulţimi mai mari U numită ***univers***. De exemplu, dacă luăm în considerare diferite mulţimi formate numai din întregi, mulţimea $\mathbb{Z}$ este un univers potrivit. Dându-se un univers U, definim ***complementul*** unei mulţimi A ca fiind $\overline{A} = U - A$. Pentru orice mulţime $A \subseteq U$, următoarele propoziţii sunt adevărate:

$$\begin{aligned} \overline{\overline{A}} &= A, \\ A \cap \overline{A} &= \emptyset, \\ A \cup \overline{A} &= U. \end{aligned}$$

Dându-se două mulţimi $A, B \subseteq U$, regulile lui DeMorgan pot fi rescrise cu ajutorul complementelor mulţimilor A şi B faţă de U astfel:

$$\begin{aligned} \overline{A \cap B} &= \overline{A} \cup \overline{B}, \\ \overline{A \cup B} &= \overline{A} \cap \overline{B}. \end{aligned}$$

Două mulţimi A şi B sunt ***disjuncte*** dacă nu au nici un element comun, adică $A \cap B = \emptyset$. O colecţie $\mathcal{S} = \{S_i\}$ de mulţimi nevide formează o ***partiţie*** a unei mulţimi S dacă:

- mulţimile sunt ***distincte două câte două***, adică $S_i, S_j \in \mathcal{S}$ şi $i \neq j$ implică $S_i \cap S_j = \emptyset$ şi
- reuniunea lor este S, adică

$$S = \bigcup_{S_i \in S.} S_i$$

Cu alte cuvinte, $\mathcal{S}$ formează o partiţie a lui S dacă fiecare element al mulţimii S apare în exact o mulţime $S_i \in \mathcal{S}$.

Numărul de elemente ale unei mulţimi poartă denumirea de ***cardinal*** (sau ***dimensiune***) al mulţimii şi este notat cu $|S|$. Două mulţimi au acelaşi cardinal dacă poate fi stabilită o corespondenţă biunivocă între cele două mulţimi. Cardinalul mulţimii vide este $|\emptyset| = 0$. În cazul în care cardinalul unei mulţimi este un număr natural, spunem că mulţimea este ***finită***; altfel, ea este ***infinită***. O mulţime infinită care poate fi pusă în corespondenţă biunivocă cu mulţimea numerelor naturale $\mathbb{N}$ este numită ***mulţime numărabilă infinită***; în caz contrar

ea este ***nenumărabilă***. Mulţimea $\mathbb{Z}$ a numerelor întregi este numărabilă dar mulţimea $\mathbb{R}$ a numerelor reale este nenumărabilă.

Pentru oricare două mulţimi finite A şi B, avem identitatea:

$$|A \cup B| = |A| + |B| - |A \cap B|, \tag{5.3}$$

de unde putem deduce că

$$|A \cup B| \leq |A| + |B|.$$

Dacă A şi B sunt disjuncte, atunci $|A \cap B| = 0$, deci $|A \cup B| = |A| + |B|$. Dacă $A \subseteq B$, atunci $|A| \leq |B|$.

O mulţime finită cu n elemente este uneori denumită ***n-mulţime***. O 1-mulţime este denumită ***mulţime cu un singur element***. O submulţime cu k elemente a unei mulţimi este denumită uneori ***k-submulţime***.

Mulţimea tuturor submulţimilor unei mulţimi S, incluzând mulţimea vidă şi mulţimea S, este notată prin 2^S şi este denumită ***mulţimea părţilor*** lui S. De exemplu, $2^{\{a,b\}} = \{\emptyset, \{a\}, \{b\}, \{a,b\}\}$. Mulţimea părţilor unei mulţimi finite S are cardinalul $2^{|S|}$.

Uneori ne interesează structuri asemănătoare mulţimilor în cadrul cărora elementele sunt ordonate. O ***pereche ordonată*** formată din două elemente a şi b este notată cu (a, b) şi poate fi definită ca fiind mulţimea $(a, b) = \{a, \{a, b\}\}$. Deci perechea ordonată (a, b) *nu* este identică cu perechea ordonată (b, a).

Produsul cartezian a două mulţimi A şi B, notat prin $A \times B$, este mulţimea tuturor perechilor ordonate astfel încât primul element al perechii face parte din mulţimea A, iar al doilea este un element al mulţimii B. Mai exact:

$$A \times B = \{(a, b) : a \in A \text{ şi } b \in B\}.$$

De exemplu $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. Când A şi B sunt mulţimi finite, cardinalul produsului lor cartezian este:

$$|A \times B| = |A| \cdot |B|. \tag{5.4}$$

Produsul cartezian a n mulţimi $A_1, A_2, \ldots, A_n$ este o mulţime de ***n-tuple***

$$A_1 \times A_2 \times \cdots \times A_n = \{(a_1, a_2, \ldots, a_n) : a_i \in A_i, i = 1, 2, \ldots, n\},$$

al cărei cardinal este:

$$|A_1 \times A_2 \times \cdots \times A_n| = |A_1| \cdot |A_2| \cdots |A_n|,$$

dacă toate mulţimile sunt finite. Notăm produsul cartezian dintre o mulţime A şi ea însăşi prin

$$A^2 = A \times A.$$

Similar, avem:

$$A^n = A \times A \times \cdots \times A.$$

Cardinalul acestei ultime mulţimi este $|A^n| = |A|^n$ dacă A este finită. Un n-tuplu poate fi privit ca un şir finit de lungime n (vezi secţiunea 5.3).

Exerciţii

5.1-1 Desenaţi diagramele Venn care ilustrează prima regulă de distributivitate (5.1).

5.1-2 Demonstraţi regulile generalizate ale lui DeMorgan pentru orice colecţie finită de mulţimi:

$$\overline{A_1 \cap A_2 \cap \cdots \cap A_n} = \overline{A_1} \cup \overline{A_2} \cup \cdots \cup \overline{A_n},$$
$$\overline{A_1 \cup A_2 \cup \cdots \cup A_n} = \overline{A_1} \cap \overline{A_2} \cap \cdots \cap \overline{A_n}.$$

5.1-3 ⋆ Demonstraţi generalizarea ecuaţiei (5.3), care este denumită ***principiul includerii şi al excluderii***:

$$\begin{aligned}
&|A_1 \cup A_2 \cup \cdots \cup A_n| = \\
&\quad |A_1| + |A_2| + \cdots + |A_n| - \\
&\quad -|A_1 \cap A_2| - |A_1 \cap A_3| - \cdots && \text{(toate perechile)} \\
&\quad +|A_1 \cap A_2 \cap A_3| + \cdots && \text{(toate tripletele)} \\
&\qquad\quad \vdots \\
&\quad +(-1)^{n-1}|A_1 \cap A_2 \cap \cdots \cap A_n|.
\end{aligned}$$

5.1-4 Arătaţi că mulţimea numerelor naturale impare este numărabilă.

5.1-5 Arătaţi că, pentru orice mulţime finită S, mulţimea părţilor sale 2^S are $2^{|S|}$ elemente (adică există $2^{|S|}$ submulţimi distincte ale lui S).

5.1-6 Daţi o definiţie inductivă pentru un n-tuplu extinzând definiţia perechii ordonate din teoria mulţimilor.

5.2. Relaţii

O ***relaţie binară*** R între două mulţimi A şi B este o submulţime a produsului cartezian $A \times B$. Dacă $(a, b) \in R$, folosim uneori notaţia $a\,R\,b$. Când spunem că R este o relaţie binară peste o mulţime A, înţelegem că R este o submulţime a produsului cartezian $A \times A$. De exemplu, relaţia "mai mic decât" peste mulţimea numerelor naturale este mulţimea $\{(a, b) : a, b \in \mathbb{N} \text{ şi } a < b\}$. O relaţie n-ară între mulţimile $A_1, A_2, \ldots, A_n$ este o submulţime a lui $A_1 \times A_2 \times \cdots \times A_n$.

O relaţie binară $R \subseteq A \times A$ este ***reflexivă*** dacă

$$a\,R\,a$$

pentru orice $a \in A$. De exemplu, "$=$" şi "$\leq$" sunt relaţii reflexive peste $\mathbb{N}$ în timp ce relaţia "$<$" nu este. Relaţia R este ***simetrică*** dacă

$$a\,R\,b \text{ implică } b\,R\,a$$

pentru orice $a, b \in A$. De exemplu, "$=$" este simetrică, dar "$<$" şi "$\leq$" nu sunt. Relaţia R este ***tranzitivă*** dacă

$$a\,R\,b \text{ şi } b\,R\,c \text{ implică } a\,R\,c$$

pentru orice $a, b, c \in A$. De exemplu, relaţiile "$<$", "$\leq$" şi "$=$" sunt tranzitive, dar relaţia $R = \{(a, b) : a, b \in \mathbb{N}$ şi $a = b - 1\}$ nu este deoarece $3\,R\,4$ şi $4\,R\,5$ nu implică $3\,R\,5$.

O relaţie care este reflexivă, simetrică şi tranzitivă poartă denumirea de ***relaţie de echivalenţă***. De exemplu "$=$" este o relaţie de echivalenţă peste mulţimea numerelor naturale în timp ce "$<$" nu este. Dacă R este o relaţie de echivalenţă peste o mulţime A, atunci pentru oricare $a \in A$, ***clasa de echivalenţă*** a lui a este mulţimea $[a] = \{b \in A : a\ R\ b\}$, adică mulţimea tuturor elementelor echivalente cu a. De exemplu, dacă definim $R = \{(a, b) : a, b \in \mathbb{N}$ şi $a + b$ este un număr par$\}$, atunci R este o relaţie de echivalenţă deoarece $a + a$ este par (reflexivitate), dacă $a + b$ este par atunci şi $b + a$ este par (simetrie) şi, dacă $a + b$ este par şi $b + c$ este par, atunci şi $a + c$ este par (tranzitivitate). Clasa de echivalenţă a lui 4 este $[4] = \{0, 2, 4, 6, \ldots\}$, iar clasa de echivalenţă a lui 3 este $[3] = \{1, 3, 5, 7, \ldots\}$. O teoremă de bază în studiul claselor de echivalenţă este următoarea.

Teorema 5.1 (O relaţie de echivalenţă este identică cu o partiţie) Clasele de echivalenţă ale oricărei relaţii de echivalenţă R peste o mulţime A formează o partiţie a lui A şi oricare partiţie a lui A determină a relaţie de echivalenţă peste A pentru care mulţimile din partiţie sunt clase de echivalenţă.

Demonstraţie. Pentru prima parte a demonstraţiei, trebuie să arătăm că clasele de echivalenţă ale lui R sunt nevide, disjuncte două câte două şi reuniunea lor este A. Deoarece R este reflexivă, $a \in [a]$, clasele de echivalenţă sunt nevide; mai mult, deoarece fiecare element $a \in A$ aparţine clasei de echivalenţă $[a]$, reuniunea claselor de echivalenţă este A. Rămâne de arătat că clasele de echivalenţă sunt distincte două câte două, adică dacă două clase de echivalenţă $[a]$ şi $[b]$ au un element c comun, ele sunt, de fapt, identice. Avem $a\,R\,c$ şi $b\,R\,c$ de unde deducem (datorită proprietăţilor de simetrie şi tranzitivitate ale lui R) că $a\,R\,b$. Deci, pentru oricare element $x \in [a]$ avem $x\,R\,a$ implică $x\,R\,b$, deci $[a] \subseteq [b]$. Similar, putem demonstra că $[b] \subseteq [a]$, deci $[a] = [b]$.

Pentru a doua parte a demonstraţiei, fie $\mathcal{A} = \{\mathcal{A}_i\}$ o partiţie a mulţimii A. Definim relaţia $R = \{(a, b) :$ există i astfel încât $a \in A_i$ şi $b \in A_i\}$. Susţinem că R este o relaţie de echivalenţă peste A. Proprietatea de reflexivitate este respectată deoarece $a \in A_i$ implică $a\,R\,a$. Proprietatea de simetrie este şi ea respectată deoarece dacă $a\,R\,b$ atunci a şi b fac parte din aceeaşi mulţime A_i, deci $b\,R\,a$. Dacă $a\,R\,b$ şi $b\,R\,c$ atunci toate cele trei elemente se află în aceeaşi mulţime, deci $b\,R\,a$ şi proprietatea de tranzitivitate este respectată. Pentru a vedea că mulţimile partiţiei sunt clase de echivalenţă ale lui R, observaţi că, dacă $a \in A_i$, atunci $x \in [a]$ implică $x \in A_i$ şi $x \in A_i$ implică $x \in [a]$. ∎

O relaţie binară R peste o mulţime A este ***antisimetrică*** dacă

$a\ R\ b$ şi $b\ R\ a$ implică $a = b$.

De exemplu, relaţia "$\leq$" peste mulţimea numerelor naturale este antisimetrică deoarece $a \leq b$ şi $b \leq a$ implică $a = b$. O relaţie care este reflexivă, antisimetrică şi tranzitivă este o ***relaţie de ordine parţială***. O mulţime peste care este definită o relaţie de ordine parţială poartă denumirea de ***mulţime parţial ordonată***. De exemplu, relaţia "este descendent al lui" este o relaţie de ordine parţială a mulţimii tuturor fiinţelor umane (dacă privim indivizii ca fiind propriii lor descendenţi).

Într-o mulţime parţial ordonată A, poate să nu existe un singur element "maxim" x, astfel încât $y\,R\,x$ pentru orice $y \in A$. Pot exista mai multe elemente ***maximale*** x, astfel încât pentru nici un $y \in A$ nu există relaţia $x\,R\,y$. De exemplu, într-o colecţie de cutii având diferite dimensiuni

pot exista mai multe cutii maximale care nu pot fi introduse în interiorul nici unei alte cutii, neexistând totuşi nici o cutie "maximă" în interiorul căreia să poată fi introduse toate celelalte cutii.

O relaţie parţială de ordine R peste o mulţime A este o ***relaţie de ordine totală*** sau ***relaţie de ordine liniară*** dacă pentru orice $a, b \in A$ avem $a\,R\,b$ sau $b\,R\,a$, adică orice pereche de elemente din A poate fi pusă în relaţie de către R. De exemplu, relaţia "$\leq$" este o relaţie de ordine totală peste mulţimea numerelor naturale, dar relaţia "este descendent al lui" nu este o relaţie de ordine totală peste mulţimea tuturor fiinţelor umane deoarece există perechi de indivizi pentru care nici unul nu este descendentul celuilalt.

Exerciţii

5.2-1 Demonstraţi că relaţia de incluziune "$\subseteq$" peste toate submulţimile lui $\mathbb{Z}$ este o relaţie de ordine parţială, dar nu este o relaţie de ordine totală.

5.2-2 Arătaţi că, pentru orice întreg pozitiv n, relaţia de "echivalenţă modulo n" este o relaţie de echivalenţă peste mulţimea numerelor întregi. (Spunem că $a \equiv b \pmod{n}$ dacă există un număr întreg q astfel încât $a - b = qn$.) În ce clase de echivalenţă partiţionează această relaţie mulţimea numerelor întregi?

5.2-3 Daţi exemple de relaţii care sunt:

a. reflexive şi simetrice, dar netranzitive,

b. reflexive şi tranzitive, dar nesimetrice,

c. simetrice şi tranzitive, dar nereflexive.

5.2-4 Fie S o mulţime finită şi R o relaţie de echivalenţă peste $S \times S$. Arătaţi că, dacă R este şi antisimetrică, atunci clasele de echivalenţă ale lui S faţă de R sunt mulţimi unitate.

5.2-5 Profesorul Narcissus susţine că, dacă o relaţie este simetrică şi tranzitivă, ea este şi reflexivă. El oferă următoarea demonstraţie: prin simetrie $a\,R\,b$ implică $b\,R\,a$, deci tranzitivitatea implică $a\,R\,a$. Are profesorul dreptate?

5.3. Funcţii

Dându-se două mulţimi A şi B, o ***funcţie*** este o relaţie binară peste $A \times B$ astfel încât, pentru orice $a \in A$, există exact un $b \in B$, şi $(a, b) \in f$. Mulţimea A este numită ***domeniul*** lui f, iar mulţimea B este numită ***codomeniul*** lui f. Uneori, scriem $f : A \to B$ şi, dacă $(a, b) \in f$, scriem $b = f(a)$, deoarece b este unic determinat prin alegerea lui a.

Intuitiv, funcţia f atribuie fiecărui element din A un element din B. Nici unui element din A nu i se atribuie două elemente diferite din B, dar acelaşi element din B poate fi atribuit mai multor elemente diferite din A. De exemplu, relaţia binară

$$f = \{(a, b) : a \in \mathbb{N} \text{ şi } b = a \bmod 2\}$$

este o funcţie $f : \mathbb{N} \to \{0, 1\}$, deoarece pentru fiecare număr natural a există exact o valoare b în $\{0, 1\}$ astfel încât $b = a \bmod 2$. Pentru acest exemplu avem $0 = f(0)$, $1 = f(1)$, $0 = f(2)$ etc. În schimb, relaţia binară

$$g = \{(a, b) : a \in \mathbb{N} \text{ şi } a + b \text{ este par}\}$$

nu este o funcţie deoarece atât $(1, 3)$ cât şi $(1, 5)$ se află în g, deci dacă alegem $a = 1$ nu putem determina un singur b pentru care $(a, b) \in g$.

Dându-se o funcţie $f : A \to B$ dacă $b = f(a)$, spunem că a este ***argumentul*** lui f, iar b este ***valoarea*** lui f în punctul a. Putem defini o funcţie enumerând valorile sale pentru fiecare element din domeniu. De exemplu, putem defini $f(n) = 2n$ pentru $n \in \mathbb{N}$, ceea ce înseamnă $f = \{(n, 2n) : n \in \mathbb{N}\}$. Două funcţii f şi g sunt ***egale*** dacă au acelaşi domeniu, acelaşi codomeniu şi, pentru orice a din domeniu, avem $f(a) = g(a)$.

Un ***şir finit*** de lungime n este o funcţie f al cărei domeniu este mulţimea $\{0, 1, \ldots, n - 1\}$. Deseori, definim un şir finit prin enumerarea valorilor sale: $\langle f(0), f(1), \ldots, f(n - 1)\rangle$. Un ***şir infinit*** este o funcţie al cărei domeniu este mulţimea numerelor naturale $\mathbb{N}$. De exemplu, şirul lui Fibonacci, definit prin (2.13), este şirul infinit $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

Când domeniul unei funcţii f este un produs cartezian, omitem adesea perechea suplimentară de paranteze în care este cuprins argumentul lui f. De exemplu, dacă $f : A_1 \times A_2 \times \cdots \times A_n \to B$, vom scrie $b = f(a_1, a_2, \ldots, a_n)$ în loc de a scrie $b = f((a_1, a_2, \ldots, a_n))$. De asemenea, vom numi fiecare a_i un ***argument*** al lui f, cu toate că singurul argument "veritabil" al lui f este n-tuplul $(a_1, a_2, \ldots, a_n)$.

Dacă $f : A \to B$ este o funcţie şi $b = f(a)$, uneori, spunem că b este ***imaginea*** lui a prin f. Imaginea unei mulţimi $A' \subseteq A$ prin f este definită prin:

$$f(A') = \{b \in B : b = f(a) \text{ pentru } a \in A'\}.$$

Domeniul de valori al unei funcţii este imaginea domeniului său, adică $f(A)$. De exemplu imaginea funcţiei $f : \mathbb{N} \to \mathbb{N}$ definită prin $f(n) = 2n$ este $f(\mathbb{N}) = \{m : m = 2n \text{ pentru } n \in \mathbb{N}\}$.

O funcţie este o ***surjecţie*** dacă imaginea sa este egală cu codomeniul său. De exemplu, funcţia $f(n) = \lfloor n/2 \rfloor$ este o funcţie surjectivă de la $\mathbb{N}$ la $\mathbb{N}$ deoarece fiecare element din $\mathbb{N}$ este valoare a lui f pentru un anumit argument. În schimb, funcţia $f(n) = 2n$ nu este o funcţie surjectivă de la $\mathbb{N}$ la $\mathbb{N}$ deoarece pentru nici un argument funcţia nu ia valoarea 3. Funcţia $f(n) = 2n$ este, totuşi, o funcţie surjectivă de la mulţimea numerelor naturale la mulţimea numerelor pare.

O funcţie $f : A \to B$ este o ***injecţie*** dacă, pentru argumente distincte, avem valori distincte ale funcţiei, adică $a \neq a'$ implică $f(a) \neq f(a')$. De exemplu funcţia $f(n) = 2n$ este o funcţie injectivă de la $\mathbb{N}$ la $\mathbb{N}$ deoarece fiecare număr par b este imaginea prin f a cel mult un element din domeniu, mai precis a lui $b/2$. Funcţia $f(n) = \lfloor n/2 \rfloor$ nu este injectivă deoarece ia valoarea 1 pentru mai multe argumente: 2 şi 3. O injecţie se numeşte uneori funcţie ***unu-la-unu***.

O funcţie $f : A \to B$ este o ***bijecţie*** dacă este injectivă şi surjectivă. De exemplu, funcţia $f(n) = (-1)^n \lceil n/2 \rceil$ este o bijecţie de la $\mathbb{N}$ la $\mathbb{Z}$:

$$\begin{array}{lcl} 0 & \to & 0, \\ 1 & \to & -1, \\ 2 & \to & 1, \\ 3 & \to & -2, \\ 4 & \to & 2, \\ & \vdots & \end{array}$$

Funcţia este injectivă, deoarece nici un element din $\mathbb{Z}$ nu este imagine a mai multor elemente din $\mathbb{N}$, şi surjectivă, deoarece fiecare element din $\mathbb{Z}$ este imagine a unui element din $\mathbb{N}$. În concluzie, funcţia este bijectivă. O bijecţie este uneori numită şi ***corespondenţă unu-la-unu*** deoarece "împerechează" elemente din domeniu şi codomeniu. O bijecţie de la o mulţime A la ea însăşi este uneori numită o ***permutare***.

Când o funcţie f este bijectivă ***inversa*** ei f^{-1} este definită prin:

$f^{-1}(b) = a$ dacă şi numai dacă $f(a) = b$.

De exemplu, inversa funcţiei $f(n) = (-1)^n \lceil n/2 \rceil$ este

$$f^{-1}(m) = \begin{cases} 2m & \text{dacă } m \geq 0, \\ -2m - 1 & \text{dacă } m < 0. \end{cases}$$

Exerciţii

5.3-1 Fie A şi B două mulţimi finite şi o funcţie $f : A \to B$. Arătaţi că:

a. dacă f este injectivă atunci $|A| \leq |B|$;

b. dacă f este surjectivă atunci $|A| \geq |B|$.

5.3-2 Verificaţi dacă funcţia $f(x) = x + 1$ este bijectivă în cazul în care domeniul şi codomeniul sunt $\mathbb{N}$. Dar dacă domeniul şi codomeniul sunt $\mathbb{Z}$?

5.3-3 Daţi o definiţie naturală pentru inversa unei relaţii binare, astfel încât dacă o relaţie binară este, de fapt, o funcţie bijectivă, atunci inversa relaţiei este inversa funcţiei.

5.3-4 ⋆ Daţi o bijecţie între $\mathbb{Z}$ şi $\mathbb{Z} \times \mathbb{Z}$.

5.4. Grafuri

Această secţiune prezintă două tipuri de grafuri: orientate şi neorientate. Cititorul trebuie avizat că anumite definiţii din literatura de specialitate diferă de cele prezentate aici, dar în cele mai multe cazuri diferenţele sunt minore. Secţiunea 23.1 arată cum pot fi reprezentate grafurile în memoria calculatorului.

Un ***graf orientat*** (sau ***digraf***) G este o pereche (V, E), unde V este o mulţime finită, iar E este o relaţie binară pe V. Mulţimea V se numeşte ***mulţimea vârfurilor*** lui G, iar elementele ei se numesc ***vârfuri***. Mulţimea E se numeşte ***mulţimea arcelor*** lui G, şi elementele ei se numesc ***arce***. Figura 5.2(a) este o reprezentare grafică a unui graf orientat cu mulţimea de vârfuri $\{1, 2, 3, 4, 5, 6\}$. În figură, vârfurile sunt reprezentate prin cercuri, iar arcele prin săgeţi. Observaţi că sunt posibile ***autobuclele*** – arce de la un vârf la el însuşi.

Într-un ***graf neorientat*** $G = (V, E)$, ***mulţimea muchiilor*** E este constituită din perechi de vârfuri *neordonate*, şi nu din perechi ordonate. Cu alte cuvinte, o muchie este o mulţime $\{u, v\}$, unde $u, v \in V$ şi $u \neq v$. Prin convenţie, pentru o muchie vom folosi notaţia (u, v) în locul notaţiei pentru mulţimi $\{u, v\}$, iar (u, v) şi (v, u) sunt considerate a fi aceeaşi muchie. Într-un graf neorientat, autobuclele sunt interzise şi astfel fiecare muchie este formată din exact două

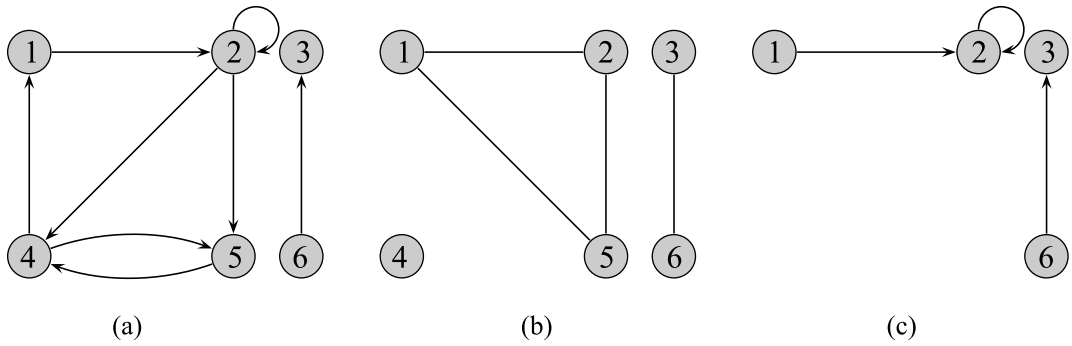


Figura 5.2 Grafuri orientate şi neorientate. **(a)** Un graf orientat $G = (V, E)$, unde $V = \{1, 2, 3, 4, 5, 6\}$ şi $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. Arcul $(2, 2)$ este o autobuclă. **(b)** Un graf neorientat $G = (V, E)$, unde $V = \{1, 2, 3, 4, 5, 6\}$ şi $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. Vârful 4 este izolat. **(c)** Subgraful grafului de la (a) indus de mulţimea de vârfuri $\{1, 2, 3, 6\}$.

vârfuri distincte. Figura 5.2(b) este o reprezentare grafică a unui graf neorientat având mulţimea de vârfuri $\{1, 2, 3, 4, 5, 6\}$.

Multe definiţii pentru grafuri orientate şi neorientate sunt aceleaşi, deşi anumiţi termeni pot avea semnificaţii diferite în cele două contexte. Dacă (u, v) este un arc într-un graf orientat $G = (V, E)$, spunem că (u, v) este ***incident din*** sau ***pleacă din*** vârful u şi este ***incident în*** sau ***intră în*** vârful v. De exemplu, arcele care pleacă din vârful 2, în figura 5.2(a), sunt $(2, 2)$, $(2, 4)$ şi $(2, 5)$. Arcele care intră în vârful 2 sunt $(1, 2)$ şi $(2, 2)$. Dacă (u, v) este o muchie într-un graf neorientat $G = (V, E)$, spunem că (u, v) este ***incidentă*** vârfurilor u şi v. În figura 5.2(b), muchiile incidente vârfului 2 sunt $(1, 2)$ şi $(2, 5)$.

Dacă (u, v) este o muchie (arc) într-un graf $G = (V, E)$, spunem că vârful v este ***adiacent*** vârfului u. Atunci când graful este neorientat, relaţia de adiacenţă este simetrică. Când graful este orientat, relaţia de adiacenţă nu este neapărat simetrică. Dacă v este adiacent vârfului u într-un graf orientat, uneori scriem $u \rightarrow v$. În părţile (a) şi (b) ale figurii 5.2, vârful 2 este adiacent vârfului 1, deoarece muchia (arcul) $(1, 2)$ aparţine ambelor grafuri. Vârful 1 *nu* este adiacent vârfului 2 în figura 5.2(a), deoarece muchia $(2, 1)$ nu aparţine grafului.

Gradul unui vârf al unui graf neorientat este numărul muchiilor incidente acestuia. De exemplu, vârful 2, din figura 5.2(b), are gradul 2. Un vârf al cărui grad este 0, cum este, de exemplu, vârful 4, din figura 5.2(b), se numeşte ***vârf izolat***. Într-un graf orientat, ***gradul exterior*** al unui vârf este numărul arcelor ce pleacă din el, iar ***gradul interior*** al unui vârf este numărul arcelor ce intră în el. ***Gradul*** unui vârf al unui graf orientat este gradul său interior, plus gradul său exterior. Vârful 2 din figura 5.2(a) are gradul interior 2, gradul exterior 3 şi gradul 5.

Un ***drum*** de ***lungime*** k de la un vârf u la un vârf u' într-un graf $G = (V, E)$ este un şir de vârfuri $\langle v_0, v_1, v_2, ..., v_k \rangle$ astfel încât $u = v_0$, $u' = v_k$, şi $(v_{i-1}, v_i) \in E$ pentru $i = 1, 2, ..., k$. Lungimea unui drum este numărul de muchii (arce) din acel drum. Drumul ***conţine*** vârfurile $v_0, v_1, v_2, ..., v_k$ şi muchiile $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. Dacă există un drum p de la u la u', spunem că u' este ***accesibil*** din u prin p, relaţie reprezentată uneori prin $u \overset{p}{\rightsquigarrow} u'$ dacă G este orientat. Un drum este ***elementar*** dacă toate vârfurile din el sunt distincte. În figura 5.2(a), drumul $\langle 1, 2, 5, 4 \rangle$ este un drum elementar de lungime 3. Drumul $\langle 2, 5, 4, 5 \rangle$ nu este elementar.

Un ***subdrum*** al unui drum $p = \langle v_0, v_1, ..., v_k \rangle$ este un subşir continuu al vârfurilor sale. Cu

alte cuvinte, pentru orice $0 \leq i \leq j \leq k$, subşirul de vârfuri $\langle v_i, v_{i+1}, ..., v_j \rangle$ este un subdrum al lui p.

Într-un graf orientat, un drum $\langle v_0, v_1, ..., v_k \rangle$ formează un ***ciclu*** dacă $v_0 = v_k$ şi drumul conţine cel puţin o muchie. Ciclul este ***elementar*** dacă, pe lângă cele de mai sus, $v_1, v_2, ..., v_k$ sunt distincte. O autobuclă este un ciclu de lungime 1. Două drumuri $\langle v_0, v_1, v_2, ..., v_{k-1}, v_0 \rangle$ şi $\langle v'_0, v'_1, v'_2, ..., v'_{k-1}, v'_0 \rangle$ formează acelaşi ciclu dacă există un număr întreg j astfel încât $v'_i = v_{(i+j) \bmod k}$ pentru $i = 0, 1, ..., k-1$. În figura 5.2(a), drumul $\langle 1, 2, 4, 1 \rangle$ formează acelaşi ciclu ca drumurile $\langle 2, 4, 1, 2 \rangle$ şi $\langle 4, 1, 2, 4 \rangle$. Acest ciclu este elementar, dar ciclul $\langle 1, 2, 4, 5, 4, 1 \rangle$ nu este elementar. Ciclul $\langle 2, 2 \rangle$ format de muchia $(2, 2)$ este o autobuclă. Un graf orientat fără autobucle este ***elementar***. Într-un graf neorientat, un drum $\langle v_0, v_1, ..., v_k \rangle$ formează un ***ciclu elementar*** dacă $k > 3$, $v_0 = v_k$ şi vârfurile $v_1, v_2, ..., v_k$ sunt distincte. De exemplu, în figura 5.2(b), drumul $\langle 1, 2, 5, 1 \rangle$ este un ciclu. Un graf fără cicluri este ***aciclic***.

Un graf neorientat este ***conex*** dacă fiecare pereche de vârfuri este conectată printr-un drum. ***Componentele conexe*** ale unui graf sunt clasele de echivalenţă ale vârfurilor sub relaţia "este accesibil din". Graful din figura 5.2(b) are trei componente conexe: $\{1, 2, 5\}$, $\{3, 6\}$, şi $\{4\}$. Fiecare vârf din $\{1, 2, 5\}$ este accesibil din fiecare vârf din $\{1, 2, 5\}$. Un graf neorientat este conex dacă are exact o componentă conexă, sau, altfel spus, dacă fiecare vârf este accesibil din fiecare vârf diferit de el.

Un graf orientat este ***tare conex*** dacă fiecare două vârfuri sunt accesibile din celălalt. ***Componentele tare conexe*** ale unui graf sunt clasele de echivalenţă ale vârfurilor sub relaţia "sunt reciproc accesibile". Un graf orientat este tare conex dacă are doar o singură componentă tare conexă. Graful din figura 5.2(a) are trei componente tare conexe: $\{1, 2, 4, 5\}$, $\{3\}$, şi $\{6\}$. Toate perechile de vârfuri din $\{1, 2, 4, 5\}$ sunt reciproc accesibile. Vârfurile $\{3, 6\}$ nu formează o componentă tare conexă, deoarece vârful 6 nu este accesibil din vârful 3.

Două grafuri $G = (V, E)$ şi $G' = (V', E')$ sunt ***izomorfe*** dacă există o bijecţie $f : V \to V'$ astfel încât $(u, v) \in E$ dacă, şi numai dacă, $(f(u), f(v)) \in E'$. Cu alte cuvinte, putem reeticheta vârfurile lui G pentru ca acestea să fie vârfuri din G', păstrând muchiile corespunzătoare din G şi G'. Figura 5.3(a) prezintă o pereche de grafuri izomorfe G şi G' cu mulţimile de vârfuri $V = \{1, 2, 3, 4, 5, 6\}$ şi $V' = \{u, v, w, x, y, z\}$. Funcţia din V în V', dată de $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$, este funcţia bijectivă cerută. Grafurile din figura 5.3(b) nu sunt izomorfe. Deşi ambele grafuri au 5 vârfuri şi 7 muchii, graful din partea superioară are un vârf cu gradul egal 4, în timp ce graful din partea inferioară a figurii nu posedă un astfel de vârf.

Spunem că un graf $G' = (V', E')$ este un ***subgraf*** al grafului $G = (V, E)$ dacă $V' \subseteq V$ şi $E' \subseteq E$. Dată fiind o mulţime $V' \subseteq V$, subgraful lui G ***indus*** de V' este graful $G' = (V', E')$, unde

$$E' = \{(u, v) \in E : u, v \in V'\}.$$

Subgraful indus de mulţimea de vârfuri $\{1, 2, 3, 6\}$, din figura 5.2(a), apare în figura 5.2(c) şi are mulţimea de muchii $\{(1, 2), (2, 2), (6, 3)\}$.

Dat fiind un graf neorientat $G = (V, E)$, ***versiunea orientată*** a lui G este graful orientat $G' = (V, E')$, unde $(u, v) \in E'$ dacă, şi numai dacă, $(u, v) \in E$. Cu alte cuvinte, fiecare muchie neorientată (u, v) din G este înlocuită în versiunea orientată prin două arce orientate: (u, v) şi (v, u). Dat fiind un graf orientat $G = (V, E)$, ***versiunea neorientată*** a lui G este graful neorientat $G' = (V, E')$, unde $(u, v) \in E'$ dacă, şi numai dacă, $u \neq v$ şi $(u, v) \in E$. Deci versiunea neorientată conţine arcele lui G "cu direcţiile eliminate" şi cu autobuclele eliminate. (Deoarece

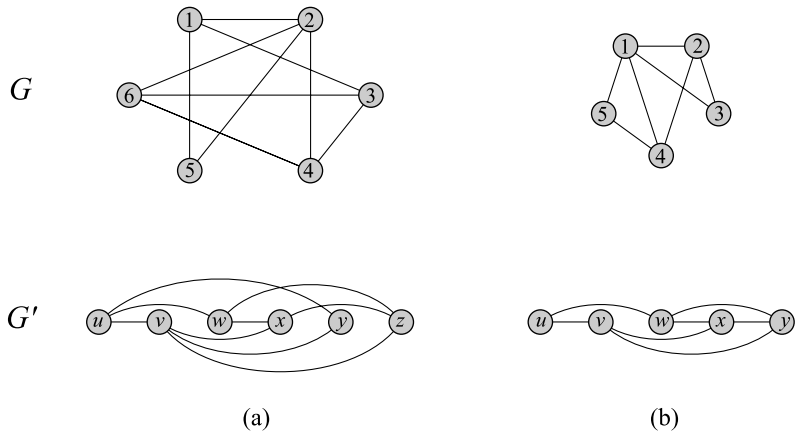


Figura 5.3 **(a)** O pereche de grafuri izomorfe. Corespondenţa dintre vârfurile grafului de sus şi cele ale grafului de jos este realizată prin funcţia dată de $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$. **(b)** Două grafuri care nu sunt izomorfe, deoarece graful de sus are un vârf de grad 4, iar graful de jos nu.

(u, v) şi (v, u) reprezintă aceeaşi muchie într-un graf neorientat, versiunea neorientată a unui graf orientat o conţine o singură dată, chiar dacă graful orientat conţine atât muchia (u, v) cât şi muchia (v, u).) Într-un graf orientat $G = (V, E)$, un ***vecin*** al unui vârf u este orice vârf care este adiacent lui u în versiunea neorientată a lui G. Deci v este un vecin al lui u dacă $(u, v) \in E$ sau $(v, u) \in E$. Într-un graf neorientat, u şi v sunt vecine dacă sunt adiacente.

Mai multe tipuri de grafuri poartă nume speciale. Un ***graf complet*** este un graf neorientat în care oricare două vârfuri sunt adiacente. Un ***graf bipartit*** este un graf neorientat $G = (V, E)$ în care mulţimea V poate fi partiţionată în două mulţimi V_1 şi V_2 astfel încât $(u, v) \in E$ implică fie că $u \in V_1$ şi $v \in V_2$, fie că $u \in V_2$ şi $v \in V_1$. Cu alte cuvinte, toate muchiile merg de la mulţimea V_1 la mulţimea V_2 sau invers. Un graf neorientat, aciclic este o ***pădure***, iar un graf neorientat, aciclic şi conex este un ***arbore (liber)*** (vezi secţiunea 5.5).

Există două variante de grafuri care pot fi întâlnite ocazional. Un ***multigraf*** seamănă cu un graf neorientat, însă poate avea atât muchii multiple între vârfuri cât şi autobucle. Un ***hipergraf*** seamănă, de asemenea, cu un graf neorientat, dar fiecare ***hipermuchie***, în loc de a conecta două vârfuri, conectează o submulţime arbitrară de vârfuri. Mulţi algoritmi scrişi pentru grafuri orientate şi neorientate obişnuite pot fi adaptaţi pentru a rula pe aceste structuri asemănătoare grafurilor.

Exerciţii

5.4-1 Invitaţii unei petreceri studenţeşti îşi strâng mâinile când se salută unul pe celălalt, şi fiecare profesor îşi aminteşte de câte ori a salutat pe cineva. La sfârşitul petrecerii, decanul facultăţii însumează numărul strângerilor de mână făcute de fiecare profesor. Arătaţi că rezultatul este par demonstrând ***lema strângerilor de mână***: dacă $G = (V, E)$ este un graf neorientat,

atunci

$$\sum_{v \in V} \text{grad}(v) = 2|E|.$$

5.4-2 Arătaţi că, dacă un graf orientat sau neorientat conţine un drum între două vârfuri u şi v, atunci conţine un drum elementar între u şi v. Arătaţi că dacă un graf orientat conţine un ciclu, atunci conţine şi un ciclu elementar.

5.4-3 Arătaţi că orice graf neorientat, conex $G = (V, E)$ satisface relaţia $|E| \geq |V| - 1$.

5.4-4 Verificaţi faptul că, într-un graf neorientat, relaţia "este accesibil din" este o relaţie de echivalenţă pe vârfurile grafului. Care din cele trei proprietăţi ale unei relaţii de echivalenţă sunt în general adevărate pentru relaţia "este accesibil din" pe vârfurile unui graf orientat?

5.4-5 Care este versiunea neorientată a grafului orientat din figura 5.2(a)? Care este versiunea orientată a grafului neorientat din figura 5.2(b)?

5.4-6 ⋆ Arătaţi că un hipergraf poate fi reprezentat printr-un graf bipartit dacă stabilim ca incidenţa în hipergraf să corespundă adiacenţei în graful bipartit. (*Indicaţie:* Presupuneţi că o mulţime de vârfuri din graful bipartit corespunde vârfurilor din hipergraf, iar cealaltă mulţime de vârfuri a grafului bipartit corespunde hipermuchiilor.)

5.5. Arbori

La fel ca în cazul grafurilor, există multe noţiuni de arbori înrudite, dar uşor diferite. Această secţiune prezintă definiţii şi proprietăţi matematice pentru mai multe tipuri de arbori. Secţiunile 11.4 şi 23.1 descriu modurile de reprezentare a arborilor în memoria calculatorului.

5.5.1. Arbori liberi

După cum a fost definit în secţiunea 5.4, un ***arbore liber*** este un graf neorientat, aciclic şi conex. Deseori omitem adjectivul "liber" atunci când spunem că un graf este un arbore. Dacă un graf neorientat este aciclic, dar s-ar putea să nu fie conex, el formează o ***pădure***. Mulţi algoritmi pentru arbori funcţionează, de asemenea, şi pe păduri. Figura 5.4(a) prezintă un arbore liber, iar figura 5.4(b) prezintă o pădure. Pădurea din figura 5.4(b) nu este un arbore pentru că nu este conexă. Graful din figura 5.4(c) nu este nici arbore şi nici pădure, deoarece conţine un ciclu.

Următoarea teoremă prezintă într-o formă concentrată multe proprietaţi importante ale arborilor liberi.

Teorema 5.2 (Proprietăţile arborilor liberi) Fie $G = (V, E)$ un graf neorientat. Următoarele afirmaţii sunt adevărate:

1. G este un arbore liber.
2. Oricare două vârfuri din G sunt conectate printr-un drum elementar unic.

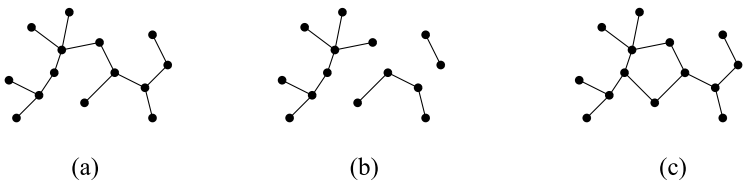


Figura 5.4 **(a)** Un arbore liber. **(b)** O pădure. **(c)** Un graf ce conţine un ciclu, motiv pentru care nu este nici arbore, nici pădure.

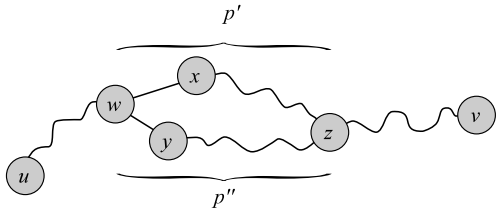


Figura 5.5 Un pas în demonstraţia teoremei 5.2. Dacă (1) G este un arbore liber, atunci (2) oricare două vârfuri din G sunt conectate printr-un unic drum elementar. Să presupunem, prin absurd, că vârfurile u şi v sunt conectate prin două drumuri simple distincte p_1 şi p_2. Aceste drumuri se despart pentru prima dată în vârful w, şi se reîntâlnesc pentru prima oară în vârful z. Drumul p' împreună cu inversul drumului p'' formează un ciclu, de unde rezultă o contradicţie.

3. G este conex, dar, dacă eliminăm o muchie oarecare din E, graful obţinut nu mai este conex.

4. G este conex, şi $|E| = |V| - 1$.

5. G este aciclic, şi $|E| = |V| - 1$.

6. G este aciclic, dar, dacă adăugăm o muchie oarecare în E, graful obţinut conţine un ciclu.

Demonstraţie. (1) $\Rightarrow$ (2): Deoarece un arbore este conex, oricare două vârfuri din G sunt conectate prin cel puţin un drum elementar. Fie u şi v două vârfuri care sunt conectate prin două drumuri distincte p_1 şi p_2, după cum este prezentat în figura 5.5. Fie w vârful unde drumurile se despart pentru prima dată. Cu alte cuvinte, w este primul vârf, atât din p_1 cât şi din p_2, al cărui succesor în p_1 este x şi al cărui succesor din p_2 este y, cu $x \neq y$. Fie z primul vârf unde drumurile se reîntâlnesc, adică z este primul vârf de după w din p_1 care se află de asemenea şi în p_2. Fie p' subdrumul din p_1 de la w la z şi care trece prin x, şi fie p'' drumul din p_2 de la w la z şi care trece prin y. Drumurile p' şi p'' nu au nici un vârf comun cu excepţia vârfurilor lor terminale. Astfel, drumul obţinut, alăturând lui p' inversul lui p'', este un ciclu. Aceasta este o contradicţie şi deci, dacă G este un arbore, nu poate exista decât cel mult un drum elementar între două vârfuri.

(2) $\Rightarrow$ (3): Dacă oricare două vârfuri din G sunt conectate printr-un drum elementar, atunci G este conex. Fie (u, v) o muchie oarecare din E. Această muchie este un drum de la u la v, şi, deci, trebuie să fie drumul unic dintre u şi v. Dacă eliminăm muchia (u, v) din G, nu mai există nici un drum între u şi v şi, astfel, eliminarea ei face ca G să nu mai fie conex.

(3) ⇒ (4): Din ipoteză, graful G este conex, iar din exerciţiul 5.4-3 avem $|E| \geq |V| - 1$. Vom demonstra relaţia $|E| \leq |V| - 1$ prin inducţie. Un graf conex cu $n = 1$ sau $n = 2$ vârfuri are $n - 1$ muchii. Să presupunem că G are $n \geq 3$ vârfuri şi că toate grafurile ce satisfac relaţia (3) şi au mai puţin de n vârfuri, satisfac, de asemenea, şi relaţia $|E| \leq |V| - 1$. Eliminând o muchie arbitrară din G, separăm graful în $k \geq 2$ componente conexe. (De fapt k este exact 2). Fiecare componentă satisface (3), deoarece, altfel, G nu ar satisface relaţia (3). Astfel, prin inducţie, dacă adunăm numărul muchiilor din fiecare componentă, obţinem cel mult $|V| - k \leq |V| - 2$. Adunând muchia eliminată, obţinem $|E| \leq |V| - 1$.

(4) ⇒ (5): Să presupunem că G este conex şi că $|E| = |V| - 1$. Trebuie să arătăm că G este aciclic. Să presupunem că G posedă un ciclu ce conţine k vârfuri $v_1, v_2, ..., v_k$. Fie $G_k = (V_k, E_k)$ subgraful lui G format din ciclul respectiv. Observaţi că $|V_k| = |E_k| = k$. Dacă $k < |V|$, atunci, trebuie să existe un vârf $v_{k+1} \in V - V_k$ care să fie adiacent unui vârf oarecare $v_i \in V_k$, din moment ce G este conex. Definim $G_{k+1} = (V_{k+1}, E_{k+1})$ ca fiind subgraful lui G având $V_{k+1} = V_k \cup \{v_{k+1}\}$ şi $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Observaţi că $|V_{k+1}| = |E_{k+1}| = k+1$. Dacă $k+1 < n$ putem continua, definind G_{k+2} în aceeaşi manieră, şi aşa mai departe, până când obţinem $G_n = (V_n, E_n)$, unde $n = |V|$, $V_n = V$, şi $|E_n| = |V_n| = |V|$. Deoarece G_n este un subgraf al lui G, avem $E_n \subseteq E$ şi de aici $|E| \geq |V|$, ceea ce contrazice presupunerea că $|E| = |V| - 1$. Astfel, G este aciclic.

(5) ⇒ (6): Să presupunem că G este aciclic şi că $|E| = |V| - 1$. Fie k numărul componentelor conexe ale lui G. Fiecare componentă conexă este, prin definiţie, un arbore liber şi, deoarece (1) implică (5), suma tuturor muchiilor din toate componentele conexe ale lui G este $|V| - k$. Prin urmare, trebuie să avem $k = 1$ şi G este de fapt un arbore. Deoarece (1) implică (2), oricare două vârfuri din G sunt conectate printr-un unic drum elementar. Astfel, adăugarea oricărei muchii la G formează un ciclu.

(6) ⇒ (1): Să presupunem că G este aciclic, dar că, dacă adăugăm o muchie arbitrară la E, creăm un ciclu. Trebuie să arătăm că G este conex. Fie u şi v două vârfuri arbitrare din G. Dacă u şi v nu sunt deja adiacente, adăugând muchia (u, v), creăm un ciclu în care toate muchiile, cu excepţia muchiei (u, v), aparţin lui G. Astfel, există un drum de la u la v şi, deoarece u şi v au fost alese arbitrar, G este conex. ∎

5.5.2. Arbori cu rădăcină şi arbori ordonaţi

Un ***arbore cu rădăcină*** este un arbore liber în care unul dintre vârfuri se deosebeşte de celelalte. Vârful evidenţiat se numeşte ***rădăcina*** arborelui. De multe ori ne referim la un vârf al unui arbore cu rădăcină ca la un ***nod***[2] al arborelui. Figura 5.6(a) prezintă un arbore cu radăcină pe o mulţime de 12 noduri a cărui rădăcină este 7.

Să luăm un nod x într-un arbore T care are rădăcina r. Orice nod y pe drumul unic din r în x este numit un ***strămoş*** al lui x. Dacă y este un strămoş al lui x, atunci x este un ***descendent*** al lui y. (Fiecare nod este atât un strămoş cât şi un descendent al lui însuşi.) Dacă y este un strămoş al lui x şi $x \neq y$, atunci y este un ***strămoş propriu*** al lui x, iar x este un ***descendent propriu*** al lui y. ***Subarborele cu rădăcina*** x este arborele indus de către descendenţii lui x şi având rădăcina x. De exemplu, subarborele având ca rădăcină nodul 8, din figura 5.6, conţine nodurile 8, 6, 5, şi 9.

Dacă ultima muchie de pe drumul de la rădăcina r a unui arbore T până la un nod x este (y, x), atunci y este ***părintele*** lui x, iar x este un ***copil*** al lui y. Rădăcina este singurul nod din

[2]Termenul de "nod" este adesea folosit în literatura de teoria grafurilor ca un sinonim pentru "vârf". Vom rezerva folosirea termenului de "nod" pentru a desemna un vârf al unui arbore cu rădăcină.

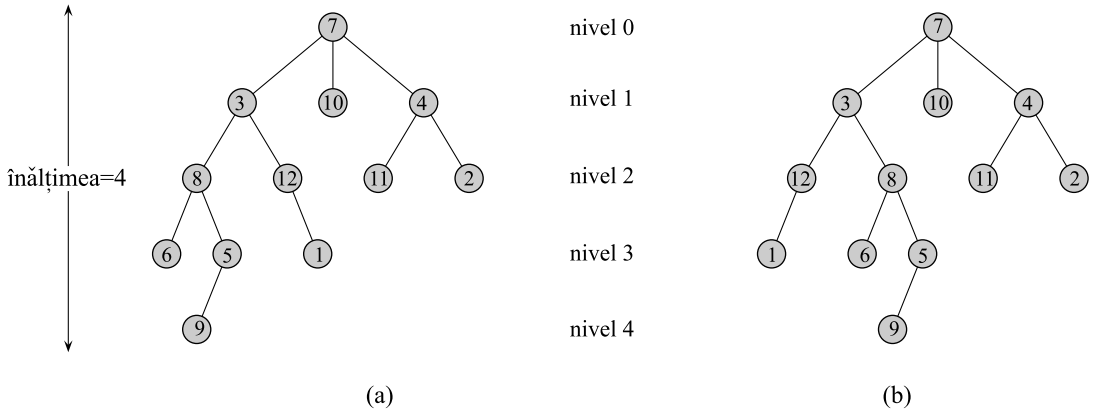


Figura 5.6 Arbori cu rădăcină şi arbori ordonaţi. **(a)** Un arbore cu rădăcină având înălţimea egală cu 4. Arborele este desenat într-un mod standard: rădăcina (nodul 7) se află în partea de sus, copiii ei (nodurile cu adâncimea 1) sub ea, copiii lor (nodurile cu adâncimea 2) sub aceştia şi aşa mai departe. Dacă arborele este ordonat, ordinea relativă de la stânga la dreapta a copiilor unui nod este importantă, altfel ea nu contează. **(b)** Un alt arbore cu rădăcină. Ca arbore cu rădăcină, este identic cu cel din (a), dar privit ca un arbore ordonat el este diferit, din moment ce copiii nodului 3 apar într-o ordine diferită.

T fără nici un părinte. Dacă două noduri au acelaşi părinte, atunci ele sunt ***fraţi***. Un nod fără nici un copil se numeşte ***nod extern*** sau ***frunză***. Un nod care nu este frunză se numeşte ***nod intern***.

Numărul copiilor unui nod x dintr-un arbore cu rădăcina T se numeşte ***gradul*** lui x.[3] Lungimea drumului de la rădăcina r la un nod x constituie ***adâncimea*** lui x în T. Cea mai mare adâncime a unui nod constituie ***înălţimea*** lui T.

Un ***arbore ordonat*** este un arbore cu rădăcină în care copiii fiecărui nod sunt ordonaţi. Cu alte cuvinte, dacă un nod are k copii, atunci există un prim copil, un al doilea copil,... şi un al k-lea copil. Cei doi arbori din figura 5.6 sunt diferiţi atunci când sunt priviţi ca fiind arbori ordonaţi, dar sunt identici atunci când sunt consideraţi a fi doar arbori cu rădăcină.

5.5.3. Arbori binari şi arbori poziţionali

Arborii binari pot fi descrişi cel mai bine într-o manieră recursivă. Un ***arbore binar*** T este o structură definită pe o mulţime finită de noduri care

- nu conţine nici un nod, sau
- este constituită din trei mulţimi de noduri disjuncte: un nod ***rădăcină***, un arbore binar numit ***subarborele stâng*** al său, şi un arbore binar numit ***subarborele drept*** al lui T.

Arborele binar care nu conţine nici un nod se numeşte ***arborele vid*** sau ***arborele nul***, notat uneori prin NIL. Dacă subarborele stâng nu este vid, atunci rădăcina acestuia se numeşte ***copilul***

[3] Observaţi că gradul unui nod depinde de modul în care este privit T: ca arbore cu rădăcină sau ca arbore liber. Gradul unui vârf dintr-un arbore liber este, ca în orice graf orientat, numărul vârfurilor adiacente lui. Cu toate acestea, într-un arbore cu rădăcină, gradul este dat de numărul de copii – părintele unui nod nu contează din perspectiva gradului.

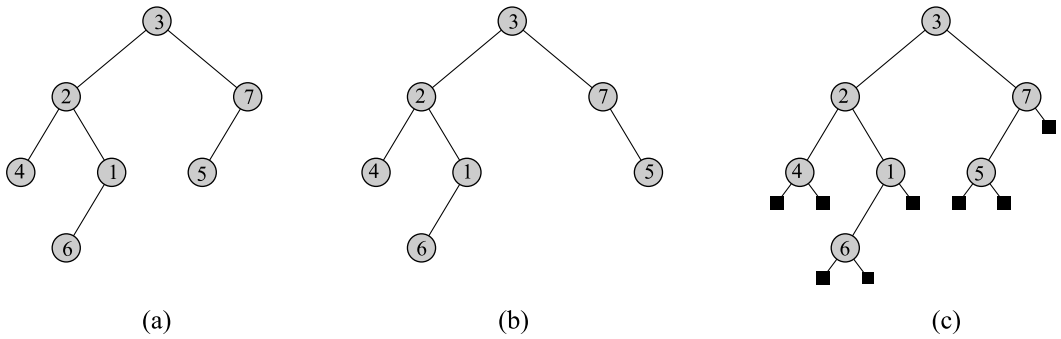


Figura 5.7 Arbori binari. **(a)** Un arbore binar desenat într-un mod standard. Copilul stâng al unui nod este desenat dedesubtul nodului şi la stânga. Copilul drept este desenat dedesubt şi la dreapta. **(b)** Un arbore binar diferit de cel din (a). În (a), copilul stâng al nodului 7 este 5, iar copilul drept este absent. În (b), copilul stâng al nodului 7 este absent, iar copilul drept este 5. Ca arbori ordonaţi, aceşti doi arbori sunt identici, dar, priviţi ca arbori binari, ei sunt diferiţi. **(c)** Arborele binar din (a) reprezentat prin nodurile interne ale unui arbore binar complet: un arbore ordonat în care fiecare nod intern are gradul 2. Frunzele din arbore sunt reprezentate prin pătrate.

stâng al rădăcinii întregului arbore. În mod asemănător, rădăcina unui subarbore drept nevid este ***copilul drept*** al rădăcinii întregului arbore. Dacă un subarbore este arborele vid NIL, copilul este ***absent*** sau ***lipseşte***. Figura 5.7(a) prezintă un arbore binar.

Un arbore binar nu este un simplu arbore ordonat în care fiecare nod are un grad mai mic sau egal cu 2. De exemplu, într-un arbore binar, dacă un nod are doar un copil, poziţia copilului – dacă este ***copilul stâng*** sau ***copilul drept*** – contează. Într-un arbore ordonat, un unic copil nu poate fi clasificat ca fiind un copil drept sau un copil stâng. Figura 5.7(b) prezintă un arbore binar care diferă de arborele din figura 5.7(a) din cauza poziţiei unui singur nod. Cu toate acestea, dacă sunt priviţi ca arbori ordonaţi, cei doi arbori sunt identici.

Informaţia poziţională dintr-un arbore binar poate fi reprezentată prin nodurile interne ale unui arbore ordonat, după cum se arată în figura 5.7(c). Ideea este să înlocuim fiecare copil absent din arborele binar cu un nod ce nu are nici un copil. Aceste noduri de tip frunză sunt reprezentate, în figură, prin pătrate. Arborele rezultat este un ***arbore binar complet***: fiecare nod este fie o frunză, fie are gradul exact 2. Nu există noduri cu gradul egal cu 1. Prin urmare, ordinea copiilor unui nod păstrează informaţia poziţională.

Informaţia poziţională care deosebeşte arborii binari de arborii ordonaţi poate fi extinsă şi la arbori cu mai mult de doi copii corespunzători unui nod. Într-un ***arbore poziţional***, copiii unui nod sunt etichetaţi cu numere întregi pozitive distincte. Al i-lea copil al unui nod este ***absent*** dacă nici un copil nu este etichetat cu numărul i. Un ***arbore k-ar*** este un arbore poziţional în care, pentru fiecare nod, toţi copiii cu etichete mai mari decât k lipsesc. Astfel, un arbore binar este un arbore k-ar cu $k = 2$.

Un ***arbore k-ar complet*** este un arbore k-ar în care toate frunzele au aceeaşi adâncime şi toate nodurile interne au gradul k. Figura 5.8 prezintă un arbore binar complet având înălţimea egală cu 3. Câte frunze are un arbore k-ar complet de înălţime h? Rădăcina are k copii de adâncime 1, fiecare dintre aceştia având k copii de adâncime 2 etc. Astfel, numărul frunzelor la adâncimea h este k^h. Prin urmare, înălţimea unui arbore k-ar complet care are n frunze este

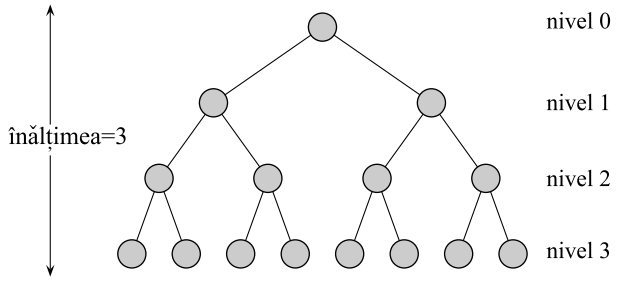


Figura 5.8 Un arbore binar complet de înălţime 3, cu 8 frunze şi 7 noduri interne.

$\log_k n$. Numărul nodurilor interne ale unui arbore k-ar complet de înălţime h este

$$1 + k + k^2 + ... + k^{h-1} = \sum_{i=0}^{h-1} k^i = \frac{k^h - 1}{k - 1}$$

din ecuaţia (3.3). Astfel, un arbore binar complet are $2^h - 1$ noduri interne.

Exerciţii

5.5-1 Desenaţi toţi arborii liberi formaţi din 3 vârfuri A, B, C. Desenaţi toţi arborii cu rădăcină având nodurile A, B, C, cu A drept rădăcină. Desenaţi toţi arborii ordonaţi având nodurile A, B, C, cu A drept rădăcină. Desenaţi toţi arborii binari având nodurile A, B, C, cu A drept rădăcină.

5.5-2 Arătaţi că, pentru $n \geq 7$, există un arbore liber cu n noduri astfel încât alegerea fiecăruia din cele n noduri drept rădăcină produce un arbore cu rădăcină diferit.

5.5-3 Fie $G = (V, E)$ un graf aciclic orientat în care există un vârf $v_0 \in V$ astfel încât există un unic drum de la v_0 la fiecare vârf $v \in V$. Demonstraţi că versiunea neorientată a lui G formează un arbore.

5.5-4 Arătaţi prin inducţie că numărul nodurilor cu gradul 2 din orice arbore binar este cu 1 mai mic decât numărul frunzelor.

5.5-5 Arătaţi, prin inducţie, că un arbore binar cu n noduri are înălţimea cel puţin egală cu $\lfloor \lg n \rfloor$.

5.5-6 ⋆ ***Lungimea drumului intern*** al unui arbore binar complet este suma adâncimilor tuturor nodurilor interne. În mod asemănător, ***lungimea drumului exterior*** este suma adâncimilor tuturor frunzelor. Se dă un arbore binar complet cu n noduri interne, având lungimea drumului interior i şi lungimea drumului exterior e. Demonstraţi că $e = i + 2n$.

5.5-7 ⋆ Asociem un cost $w(x) = 2^{-d}$ fiecărei frunze x de adâncime d dintr-un arbore binar T. Demonstraţi că $\sum_x w(x) \leq 1$, unde x este orice frunză din T. (Această relaţie este cunoscută sub numele de ***inegalitatea Kraft***.)

5.5-8 ⋆ Arătaţi că orice arbore binar cu L frunze conţine un subarbore având între $L/3$ şi $2L/3$ frunze (inclusiv).

Probleme

5-1 Colorarea grafurilor
O ***k-colorare*** a unui graf neorientat $G = (V, E)$ este o funcţie $c : V \to \{0, 1, ..., k-1\}$ astfel încât $c(u) \neq c(v)$ pentru orice muchie $(u, v) \in E$. Cu alte cuvinte, numerele $0, 1, ..., k-1$ reprezintă k culori, iar nodurile adiacente trebuie să aibă culori diferite.

a. Demonstraţi că orice arbore este 2-colorabil.

b. Arătaţi că următoarele afirmaţii sunt echivalente:

(a) G este bipartit.

(b) G este 2-colorabil.

(c) G nu are cicluri de lungime impară.

c. Fie d gradul maxim al oricărui vârf dintr-un graf G. Arătaţi că G poate fi colorat cu $d+1$ culori.

d. Arătaţi că dacă G are $O(|V|)$ muchii, atunci G poate fi colorat cu $O(\sqrt{|V|})$ culori.

5-2 Grafuri de prieteni
Reformulaţi fiecare dintre următoarele afirmaţii sub forma unei teoreme despre grafuri neorientate, iar apoi demonstraţi teorema. Puteţi presupune că prietenia este simetrică dar nu şi reflexivă.

a. În orice grup de $n \geq 2$ persoane, există două persoane cu acelaşi număr de prieteni în grup.

b. Orice grup de şase persoane conţine fie trei persoane între care există o relaţie de prietenie reciprocă, fie trei persoane care nu se cunosc nici una pe cealaltă.

c. Orice grup de persoane poate fi împărţit în două subgrupuri astfel încât cel puţin jumătate din prietenii fiecărei persoane să se afle în grupul din care persoana respectivă *nu* face parte.

d. Dacă toate persoanele dintr-un grup sunt prietene cu cel puţin jumătate din persoanele din grup, atunci grupul poate fi aşezat la o masă astfel încât fiecare persoană este aşezată între doi prieteni.

5-3 Împărţirea arborilor în două
Mulţi algoritmi divide şi stăpâneşte care lucrează pe grafuri cer ca graful să fie împarţit în două subgrafuri de dimensiuni aproximativ egale, prin eliminarea unui mic număr de muchii. Această problemă cercetează împărţirea în două a arborilor.

a. Demonstraţi că, prin eliminarea unei singure muchii, putem partiţiona vârfurile oricărui arbore binar având n vârfuri în două mulţimi A şi B astfel încât $|A| \leq 3n/4$ şi $|B| \leq 3n/4$.

b. Demonstraţi că această constantă de 3/4 de la punctul (a) este optimală în cel mai defavorabil caz, dând un exemplu de arbore simplu a cărui cea mai echilibrată partiţie, după eliminarea unei singure muchii, are proprietatea $|A| = 3n/4$.

c. Demonstraţi că, prin eliminarea a cel mult $O(\lg n)$ muchii, putem partiţiona vârfurile oricărui arbore având n vârfuri în două mulţimi A şi B astfel încât $|A| = \lfloor n/2 \rfloor$ şi $|B| = \lceil n/2 \rceil$.

Note bibliografice

G. Boole a dus o muncă de pionierat în dezvoltarea logicii simbolice şi a introdus multe dintre notaţiile de bază pentru mulţimi într-o carte publicată în 1854. Teoria modernă a mulţimilor a fost elaborată de către G. Cantor în timpul perioadei 1874–1895. Cantor s-a concentrat, în special, asupra mulţimilor cu cardinalitate infinită. Termenul de "funcţie" este atribuit lui G. W. Leibnitz, care l-a folosit pentru a se referi la mai multe feluri de formule matematice. Definiţia sa limitată a fost generalizată de multe ori. Teoria grafurilor datează din anul 1736, când L. Euler a demonstrat că este imposibil ca o persoană să traverseze fiecare din cele 7 poduri din oraşul Königsberg exact o singură dată şi să se întoarcă în locul de unde a plecat.

Un compendiu folositor, conţinând mai multe definiţii şi rezultate din teoria grafurilor, se află în cartea scrisă de Harary [94].

6 Numărare şi probabilitate

În acest capitol trecem în revistă combinatorica elementară şi teoria elementară a probabilităţilor. Dacă aveţi o bună fundamentare în aceste domenii, puteţi trece mai uşor peste începutul capitolului şi să vă concentraţi asupra secţiunilor următoare. Multe dintre capitole nu necesită probabilităţi, dar pentru anumite capitole ele sunt esenţiale.

În secţiunea 6.1 trecem în revistă principalele rezultate ale teoriei numărării (combinatorica, analiza combinatorie), inclusiv formule standard pentru numărarea permutărilor şi combinărilor. În secţiunea 6.2 sunt prezentate axiomele probabilităţii şi rezultatele de bază legate de distribuţiile de probabilitate. Variabilele aleatoare sunt introduse în secţiunea 6.3, unde se dau de asemenea proprietăţile mediei şi ale dispersiei. Secţiunea 6.4 investighează distribuţia geometrică şi cea binomială care apar în studiul probelor bernoulliene. Studiul distribuţiei binomiale este continuat în secţiunea 6.5, care este o discuţie avansată despre "cozile" distribuţiei. În fine, în secţiunea 6.6 ilustrăm analiza probabilistică prin intermediul a trei exemple: paradoxul zilei de naştere, aruncarea aleatoare a bilelor în cutii şi câştigarea liniilor (secvenţelor).

6.1. Numărare

Teoria numărării încearcă să răspundă la întrebarea "câţi", fără a număra efectiv. De exemplu, am putea să ne întrebăm "câte numere distincte de n biţi există", sau "în câte moduri pot fi ordonate n elemente distincte". În această secţiune, vom trece în revistă elementele teoriei numărării. Deoarece anumite părţi ale materialului necesită înţelegerea noţiunilor şi rezultatelor de bază referitoare la mulţimi, cititorul este sfătuit să înceapă prin a revedea materialul din secţiunea 5.1.

Regulile sumei şi produsului

O mulţime ale cărei elemente dorim să le numărăm poate fi exprimată uneori ca o reuniune de mulţimi disjuncte sau ca un produs cartezian de mulţimi.

Regula sumei ne spune că numărul de moduri de alegere a unui element din cel puţin una din două mulţimi *disjuncte* este suma cardinalilor mulţimilor. Altfel spus, dacă A şi B sunt două mulţimi finite fără nici un element comun, atunci $|A \cup B| = |A| + |B|$, ceea ce rezultă din ecuaţia (5.3). De exemplu, fiecare poziţie de pe un număr de maşină este o literă sau o cifră. Numărul de posibilităţi, pentru fiecare poziţie, este $26 + 10 = 36$ deoarece sunt 26 de alegeri pentru o literă şi 10 alegeri pentru o cifră.

Regula produsului spune că numărul de moduri în care putem alege o pereche ordonată este numărul de alegeri pentru primul element înmulţit cu numărul de alegeri pentru al doilea element. Adică, dacă A şi B sunt două mulţimi finite, atunci $|A \times B| = |A| \cdot |B|$, care este, pur şi simplu, ecuaţia (5.4). De exemplu, dacă un producător de îngheţată oferă 28 de arome de îngheţată şi 4 tipuri de glazuri, numărul total de sortimente cu un glob de îngheţată şi o glazură este $28 \cdot 4 = 112$.

Şiruri

Un ***şir*** peste o mulţime finită S este o secvenţă de elemente din S. De exemplu, există 8 şiruri binare de lungime 3:

$$000, 001, 010, 011, 100, 101, 110, 111.$$

Vom numi, uneori, ***k-şir*** un şir de lungime k. Un ***subşir*** s' al unui şir s este o secvenţă ordonată de elemente consecutive ale lui s. Un ***k-subşir*** al unui şir este un subşir de lungime k. De exemplu, 010 este un 3 subşir al lui 01101001 (3-subşirul care începe din poziţia 4), dar 111 nu este un subşir al lui 01101001.

Un k-şir peste o mulţime S poate fi privit ca un element al produsului cartezian S^k de k-tuple; astfel există $|S|^k$ şiruri de lungime k. De exemplu, numărul de k-şiruri binare este 2^k. Intuitiv, pentru a construi k-şiruri peste o mulţime de n elemente (n-mulţime), avem n moduri de alegere pentru primul element; pentru fiecare dintre aceste alegeri, avem n moduri de a alege al doilea element; şi aşa mai departe de k ori. Această construcţie ne conduce la concluzia că numărul de k-şiruri este produsul a k factori identici $n \cdot n \cdots n = n^k$.

Permutări

O ***permutare***a unei mulţimi finite S este o secvenţă ordonată a tuturor elementelor lui S, cu fiecare element apărând exact o singură dată. De exemplu, dacă $S = \{a, b, c\}$, există 6 permutări ale lui S:

$$abc, acb, bac, bca, cab, cba.$$

Există $n!$ permutări ale unei mulţimi având n elemente, deoarece primul element al secvenţei poate fi ales în n moduri, al doilea în $n-1$ moduri, al treilea în $n-2$ moduri ş.a.m.d.

O ***k-permutare***[1] a lui S este o secvenţă de k elemente ale lui S în care nici un element nu apare mai mult de o singură dată. (Astfel o permutare ordinară este o n-permutare a unei n-mulţimi.) Cele douăsprezece 2-permutări ale mulţimii $\{a, b, c, d\}$ sunt

$$ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc.$$

Numărul de k-permutări ale unei n-mulţimi este

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}, \tag{6.1}$$

deoarece există n moduri de a alege primul element, $n-1$ moduri de a alege al doilea element şi tot aşa, până se selectează k elemente, ultimul fiind o selecţie de $n-k+1$ elemente.

Combinări

O ***k-combinare*** a unei n-mulţimi S este, pur şi simplu, o k-submulţime a lui S. Există şase 2-combinări ale unei 4-mulţimi $\{a, b, c, d\}$:

$$ab, ac, ad, bc, bd, cd.$$

[1] sau ***aranjament*** – n.t.

(Aici am utilizat prescurtarea ab pentru a nota mulţimea $\{a, b\}$ ş.a.m.d.) Putem construi o k-combinare a unei n-mulţimi alegând k elemente diferite (distincte) din n-mulţime.

Numărul de k-combinări ale unei n-mulţimi poate fi exprimat cu ajutorul numărului de k-permutări ale unei n-mulţimi. Pentru orice k-combinare, există exact $k!$ permutări ale elementelor sale, fiecare fiind o k-permutare distinctă a n-mulţimii. Astfel, numărul de k-combinări ale unei n-mulţimi este numărul de k-permutări împărţit cu $k!$; din ecuaţia (6.1), această valoare este

$$\frac{n!}{k!(n-k)!}. \tag{6.2}$$

Pentru $k = 0$, această formulă ne spune că numărul de moduri în care putem alege 0 elemente dintr-o n-mulţime este 1 (nu 0), deoarece $0! = 1$.

Coeficienţi binomiali

Vom utiliza notaţia $\binom{n}{k}$ (se citeşte "combinări de n luate câte k") pentru a desemna numărul de k-combinări ale unei n-mulţimi. Din ecuaţia (6.2) avem

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{6.3}$$

Această formulă este simetrică în k şi $n - k$:

$$\binom{n}{k} = \binom{n}{n-k}. \tag{6.4}$$

Aceste numere sunt cunoscute şi sub numele de ***coeficienţi binomiali*** deoarece apar în ***dezvoltarea binomială*** (formula binomului lui Newton):

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}. \tag{6.5}$$

Un caz special al dezvoltării binomiale este acela în care $x = y = 1$:

$$2^n = \sum_{k=0}^{n} \binom{n}{k}. \tag{6.6}$$

Această formulă corespunde numărării celor 2^n n-şiruri binare după numărul de cifre 1 pe care ele îl conţin: există $\binom{n}{k}$ şiruri binare ce conţin exact k de 1 deoarece există $\binom{n}{k}$ moduri de a alege k din cele n poziţii în care vom pune cifre 1.

Există multe identităţi referitoare la coeficienţii binomiali. Exerciţiile de la sfârşitul acestei secţiuni vă oferă posibilitatea să demonstraţi câteva.

Margini binomiale

Uneori, avem nevoie să delimităm mărimea unui coeficient binomial. Pentru $1 \le k \le n$, avem marginea inferioară

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} = \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right) \ge \left(\frac{n}{k}\right)^k. \tag{6.7}$$

Utilizând inegalitatea $k! \geq (k/e)^k$, dedusă din formula lui Stirling (2.12), obţinem marginea superioară

$$\begin{aligned} \binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1} \leq \frac{n^k}{k!} \leq & (6.8) \\ &\leq \left(\frac{en}{k}\right)^k. & (6.9) \end{aligned}$$

Pentru orice $0 \leq k \leq n$, putem utiliza inducţia (vezi exerciţiul 6.1-12) pentru a demonstra delimitarea

$$\binom{n}{k} \leq \frac{n^n}{k^k (n-k)^{n-k}}, \tag{6.10}$$

unde, pentru comoditate, am presupus că $0^0 = 1$. Pentru $k = \lambda n$, unde $0 \leq \lambda \leq 1$, această margine se poate rescrie sub forma

$$\begin{aligned} \binom{n}{\lambda n} &\leq \frac{n^n}{(\lambda n)^{\lambda n}\left((1-\lambda)n\right)^{(1-\lambda)n}} = \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n & (6.11) \\ &= 2^{nH(\lambda)}, & (6.12) \end{aligned}$$

unde

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \tag{6.13}$$

este ***funcţia entropie (binară)*** şi unde, pentru comoditate, am presupus că $0 \lg 0 = 0$, astfel că $H(0) = H(1) = 0$.

Exerciţii

6.1-1 Câte k-subşiruri are un n-şir? (Considerăm k-subşirurile identice ce apar pe poziţii diferite ca fiind diferite.) Câte subşiruri are în total un n-şir?

6.1-2 O ***funcţie booleană*** având n intrări şi m ieşiri este o funcţie de la $\{\text{ADEVĂRAT, FALS}\}^n$ la $\{\text{ADEVĂRAT, FALS}\}^m$. Câte funcţii booleene având n intrări şi o ieşire există? Câte funcţii booleene având n intrări şi m ieşiri există?

6.1-3 În câte moduri se pot aşeza n profesori în jurul unei mese circulare la o conferinţă? Considerăm că două aşezări sunt identice dacă una poate fi rotită pentru a o obţine pe cealaltă.

6.1-4 În câte moduri pot fi alese trei numere distincte din mulţimea $\{1, 2, \ldots, 100\}$ astfel ca suma lor să fie pară?

6.1-5 Demonstraţi identitatea:

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{6.14}$$

pentru $0 < k \leq n$.

6.1-6 Demonstraţi identitatea:

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

pentru $0 \leq k < n$.

6.1-7 Pentru a alege k obiecte din n, puteţi marca unul dintre obiecte şi să vedeţi dacă a fost ales obiectul marcat. Utilizaţi această abordare pentru a demonstra că

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

6.1-8 Utilizând rezultatul din exerciţiul 6.1-7, construiţi o tabelă a coeficienţilor binomiali, $\binom{n}{k}$ pentru $n = 0, 1, \ldots, 6$ şi $0 \leq k \leq n$, cu $\binom{0}{0}$ în vârf, $\binom{1}{0}$ şi $\binom{1}{1}$ pe linia următoare şi aşa mai departe. O astfel de tabelă de coeficienţi binomiali se numeşte ***triunghiul lui Pascal***.

6.1-9 Demonstraţi că

$$\sum_{i=1}^{n} i = \binom{n+1}{2}.$$

6.1-10 Arătaţi că, pentru orice $n \geq 0$ şi $0 \leq k \leq n$, maximul valorii lui $\binom{n}{k}$ se realizează când $k = \lfloor n/2 \rfloor$ sau $k = \lceil n/2 \rceil$.

6.1-11 $\star$ Argumentaţi că, pentru orice $n \geq 0$, $j \geq 0$, $k \geq 0$ şi $j + k \leq n$,

$$\binom{n}{j+k} \leq \binom{n}{j}\binom{n-j}{k}. \tag{6.15}$$

Daţi atât o demonstraţie algebrică cât şi una bazată pe o metodă de alegere a celor $j+k$ elemente din n. Daţi un exemplu în care egalitatea să nu aibă loc.

6.1-12 $\star$ Utilizaţi inducţia după $k \leq n/2$ pentru a demonstra inegalitatea (6.10) şi utilizaţi ecuaţia (6.4) pentru a o extinde pentru orice $k \leq n$.

6.1-13 $\star$ Utilizaţi aproximaţia lui Stirling pentru a demonstra că

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}\left(1 + O(1/n)\right). \tag{6.16}$$

6.1-14 $\star$ Derivând funcţia entropie $H(\lambda)$, arătaţi că ea îşi atinge valoarea maximă în $\lambda = 1/2$. Cât este $H(1/2)$?

6.2. Probabilitate

Probabilitatea este un instrument esenţial pentru proiectarea şi analiza algoritmilor pur probabilişti şi a algoritmilor care se bazează pe generarea de numere aleatoare. În această secţiune, se trec în revistă bazele teoriei probabilităţilor.

Vom defini probabilitatea în termeni de ***spaţiu de selecţie*** S, care este o mulţime ale cărei elemente se numesc ***evenimente elementare***. Fiecare eveniment elementar poate fi privit ca un rezultat posibil al unui experiment. Pentru experimentul aruncării a două monede distincte şi distinctibile, putem privi spaţiul de selecţie ca fiind mulţimea tuturor 2-şirurilor peste {S, B}:

$$S = \{\text{SS}, \text{SB}, \text{BS}, \text{BB}\}.$$

Un ***eveniment*** este o submulţime[2] a spaţiului de selecţie S. De exemplu, în experimentul aruncării a două monede, evenimentul de a obţine un ban sau o stemă este {SB,BS}. Evenimentul S se numeşte ***eveniment sigur***, iar evenimentul $\emptyset$ se numeşte ***eveniment nul*** sau ***eveniment imposibil***. Spunem că două evenimente sunt ***mutual exclusive*** sau ***incompatibile*** dacă $A \cap B = \emptyset$. Vom trata, uneori, un eveniment elementar $s \in S$ ca fiind evenimentul $\{s\}$. Prin definiţie, toate evenimentele elementare sunt mutual exclusive.

Axiomele probabilităţii

O ***distribuţie de probabilitate*** (probabilitate) $\Pr\{\}$ pe spaţiul de selecţie S este o aplicaţie de la mulţimea evenimentelor lui S la mulţimea numerelor reale care satisface următoarele ***axiome ale probabilităţii***:

1. $\Pr\{A\} \geq 0$ pentru orice eveniment A.
2. $\Pr\{S\} = 1$.
3. $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ pentru oricare două evenimente mutual exclusive A şi B. Mai general, pentru orice secvenţă (finită sau infinit numărabilă) de evenimente $A_1, A_2, \ldots$ care sunt două câte două mutual exclusive

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\}.$$

Vom numi $\Pr\{A\}$ ***probabilitatea*** evenimentului A. De notat că axioma 2 este o cerinţă de normalizare: nu este nimic fundamental în alegerea lui 1 ca probabilitate a evenimentului sigur, exceptând faptul că este mai naturală şi mai convenabilă.

[2] Pentru o distribuţie de probabilitate generală, pot exista anumite submulţimi ale lui S care nu sunt considerate evenimente. Această situaţie apare de obicei când spaţiul de selecţie este infinit nenumărabil. Cerinţa principală este ca mulţimea evenimentelor din spaţiul de selecţie să fie închisă la operaţiile de complementare şi la intersecţia unui număr finit sau numărabil de evenimente. Cele mai multe dintre distribuţiile pe care le vom întâlni sunt definite peste spaţii de selecţie finite sau numărabile şi vom considera toate submulţimile spaţiului de selecţie ca fiind evenimente. O excepţie notabilă va fi distribuţia de probabilitate uniformă continuă, pe care o vom prezenta pe scurt.

Anumite rezultate se pot deduce imediat din aceste axiome şi din rezultatele de bază ale teoriei mulţimilor (vezi secţiunea 5.1). Evenimentul nul are probabilitatea $\Pr\{\emptyset\} = 0$. Dacă $A \subseteq B$, atunci $\Pr\{A\} \leq \Pr\{B\}$. Utilizând $\bar{A}$ pentru a nota evenimentul $S - A$ (***complementul*** sau ***evenimentul contrar*** lui A), avem $\Pr\{\bar{A}\} = 1 - \Pr\{A\}$. Pentru oricare două evenimente A şi B,

$$\begin{aligned} \Pr\{A \cup B\} &= \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} && (6.17) \\ &\leq \Pr\{A\} + \Pr\{B\}. && (6.18) \end{aligned}$$

În exemplul nostru cu aruncarea monedei presupunem că fiecare eveniment elementar are probabilitatea 1/4. Probabilitatea de a obţine cel puţin o stemă este

$$\Pr\{\text{SS}, \text{SB}, \text{BS}\} = \Pr\{\text{SS}\} + \Pr\{\text{SB}\} + \Pr\{\text{BS}\} = 3/4.$$

Altfel, deoarece probabilitatea de a obţine mai puţin de o stemă este $\Pr\{\text{BB}\} = 1/4$, probabilitatea de a obţine cel puţin o stemă este $1 - 1/4 = 3/4$.

Distribuţii discrete de probabilitate

O distribuţie de probabilitate este ***discretă*** dacă este definită peste un spaţiu de selecţie finit sau numărabil. Fie S spaţiul de selecţie. Atunci pentru orice eveniment A,

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\},$$

deoarece evenimentele elementare, mai concret cele din A, sunt mutual exclusive. Dacă S este finită şi fiecare eveniment elementar $s \in S$ are probabilitatea

$$\Pr\{s\} = 1/|S|,$$

atunci avem ***distribuţia de probabilitate uniformă***[3] pe S. Într-un astfel de caz, experimentul este descris adesea ca "alegerea unui element al lui S la întâmplare".

De exemplu, să considerăm procesul de aruncare a unei ***monede perfecte*** (corecte, ideale), pentru care probabilitatea de a obţine stema este aceeaşi cu probabilitatea de a obţine banul, adică 1/2. Dacă aruncăm moneda de n ori, avem distribuţia de probabilitate uniformă definită peste spaţiul de selecţie $S = \{\text{S}, \text{B}\}^n$, o mulţime de dimensiune (cardinal) 2^n. Fiecare element din S poate fi reprezentat ca un şir de lungime n peste $\{\text{S}, \text{B}\}$ şi fiecare apare cu probabilitatea $1/2^n$. Evenimentul

$$A = \{\text{stema apare de exact } k \text{ ori şi banul apare de } n - k \text{ ori}\}$$

este o submulţime a lui S, de dimensiune (cardinal) $|A| = \binom{n}{k}$ deoarece există $\binom{n}{k}$ şiruri de lungime n peste $\{\text{S}, \text{B}\}$ ce conţin exact k S-uri. Astfel probabilitatea lui A este $\Pr\{A\} = \binom{n}{k}/2^n$.

[3] distribuţia uniformă discretă pe S – n.t.

Distribuţia de probabilitate uniformă continuă

Distribuţia de probabilitate uniformă continuă este un exemplu de distribuţie de probabilitate în care nu toate submulţimile spaţiului de selecţie sunt considerate evenimente. Distribuţia de probabilitate uniformă continuă este definită pe un interval închis $[a, b]$ al axei reale, cu $a < b$. Intuitiv, dorim ca toate punctele intervalului $[a, b]$ să fie "echiprobabile". Există, totuşi, o infinitate nenumărabilă de puncte, aşa că, dacă atribuim tuturor punctelor aceeaşi probabilitate finită, pozitivă, nu putem satisface axiomele 2 şi 3. Din acest motiv, vom asocia o probabilitate numai *anumitor* submulţimi ale lui S, astfel ca axiomele să fie satisfăcute pentru astfel de evenimente.

Pentru orice interval închis $[c, d]$, unde $a \leq c \leq d \leq b$, ***distribuţia de probabilitate uniformă continuă*** defineşte probabilitatea evenimentului $[c, d]$ ca fiind

$$\Pr\{[c, d]\} = \frac{d - c}{b - a}.$$

De notat că, pentru orice punct $x = [x, x]$, probabilitatea lui x este 0. Dacă înlăturăm capetele unui interval $[c, d]$, obţinem un interval deschis (c, d). Deoarece $[c, d] = [c, c] \cup (c, d) \cup [d, d]$, pe baza axiomei 3 avem $\Pr\{[c, d]\} = \Pr\{(c, d)\}$. În general, mulţimea evenimentelor pentru distribuţia de probabilitate uniformă continuă este constituită din toate submulţimile lui $[a, b]$ care pot fi obţinute printr-o reuniune finită de intervale deschise sau închise.

Probabilitate condiţionată şi independenţă

Uneori avem, cu anticipaţie, unele cunoştinţe parţiale despre un experiment. De exemplu, să presupunem că un prieten a aruncat două monede perfecte şi că v-a spus că cel puţin una din ele a arătat stema. Care este probabilitatea să obţinem două steme? Informaţia dată elimină posibilitatea de a avea de două ori banul. Cele trei evenimente care au rămas sunt echiprobabile, deci deducem că fiecare din ele apare cu probabilitatea $1/3$. Deoarece numai unul dintre acestea înseamnă două steme, răspunsul la întrebarea noastră este $1/3$.

Probabilitatea condiţionată formalizează noţiunea de a avea cunoştinţe parţiale despre rezultatul unui experiment. ***Probabilitea condiţionată*** a evenimentului A, ştiind că un alt eveniment B a apărut, se defineşte prin

$$\Pr\{A|B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}}, \tag{6.19}$$

ori de câte ori $\Pr\{B\} \neq 0$. (Notaţia $\Pr\{A|B\}$ se va citi "probabilitatea lui A condiţionată de B".) Intuitiv, deoarece se dă că evenimentul B a apărut, evenimentul ca A să apară, de asemenea este $A \cap B$. Adică, $A \cap B$ este mulţimea rezultatelor în care apar atât A cât şi B[4]. Deoarece rezultatul este unul din evenimentele elementare ale lui B, normalizăm probabilităţile tuturor evenimentelor elementare din B, împărţindu-le la $\Pr\{B\}$, astfel ca suma lor să fie 1. Probabilitatea lui A condiţionată de B este, de aceea, raportul dintre probabilitatea evenimentului $A \cap B$ şi probabilitatea lui B. În exemplul de mai sus, A este evenimentul ca ambele monede să dea stema, iar B este evenimentul ca să se obţină cel puţin o stemă. Astfel, $\Pr\{A|B\} = (1/4)/(3/4) = 1/3$.

Două evenimente sunt ***independente*** dacă

$$\Pr\{A \cap B\} = \Pr\{A\} \Pr\{B\},$$

[4]Elementele lor sau rezultatele lor – n.t.

ceea ce este echivalent, dacă $\Pr\{B\} \neq 0$, cu condiţia

$$\Pr\{A|B\} = \Pr\{A\}.$$

De exemplu, să presupunem că se aruncă două monede perfecte şi că rezultatele sunt independente. Atunci probabilitatea de a obţine două steme este $(1/2)(1/2) = 1/4$. Presupunem acum că un eveniment este ca prima monedă să dea o stemă şi cel de-al doilea este ca monedele să dea rezultate diferite. Fiecare dintre aceste evenimente apare cu probabilitatea $1/2$, iar probabilitatea ca să apară ambele evenimente este $1/4$; astfel, conform definiţiei independenţei, evenimentele sunt independente – chiar dacă ne-am putea gândi că ambele depind de prima monedă. În fine, să presupunem că monedele sunt legate, astfel încât pe ambele să cadă fie stema, fie banul şi că ambele posibilităţi sunt egal probabile. Atunci probabilitatea ca fiecare monedă să dea stema este $1/2$, dar probabilitatea ca ambele să dea stema este $1/2 \neq (1/2)(1/2)$. În consecinţă, evenimentul ca o monedă să dea stema şi evenimentul ca cealaltă să dea stema nu sunt independente.

O colecţie de evenimente $A_1, A_2, \ldots, A_n$ se numeşte ***independentă pe perechi*** (sau spunem că evenimentele sunt ***două câte două independente*** – n.t.) dacă

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\} \Pr\{A_j\}$$

pentru orice $1 \leq i < j \leq n$. Spunem că evenimentele $A_1, \ldots, A_n$ sunt (***mutual***) ***independente*** (sau ***independente în totalitate***) dacă fiecare k-submulţime $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ a colecţiei, unde $2 \leq k \leq n$ şi $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ satisface

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \ldots \cap A_{i_k}\} = \Pr\{A_{i_1}\} \Pr\{A_{i_2}\} \ldots \Pr\{A_{i_k}\}.$$

De exemplu, să presupunem că aruncăm două monede perfecte. Fie A_1 evenimentul ca prima monedă să dea stema, fie A_2 evenimentul ca a doua monedă să dea stema şi A_3 evenimentul ca cele două monede să fie diferite. Avem

$$\begin{aligned}
\Pr\{A_1\} &= 1/2, \\
\Pr\{A_2\} &= 1/2, \\
\Pr\{A_3\} &= 1/2, \\
\Pr\{A_1 \cap A_2\} &= 1/4, \\
\Pr\{A_1 \cap A_3\} &= 1/4, \\
\Pr\{A_2 \cap A_3\} &= 1/4, \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0.
\end{aligned}$$

Deoarece pentru $1 \leq i < j \leq 3$, avem $\Pr\{A_i \cap A_j\} = \Pr\{A_i\} \Pr\{A_j\} = 1/4$, evenimentele A_1, A_2 şi A_3 sunt două câte două independente. Ele nu sunt mutual independente, deoarece $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$ şi $\Pr\{A_1\} \Pr\{A_2\} \Pr\{A_3\} = 1/8 \neq 0$.

Teorema lui Bayes

Din definiţia probabilităţii condiţionate (6.19) rezultă că, pentru două evenimente A şi B, fiecare cu probabilitate nenulă, avem

$$\Pr\{A \cap B\} = \Pr\{B\} \Pr\{A|B\} = Pr\{A\} \Pr\{B|A\} \tag{6.20}$$

Exprimând din ecuaţiile de mai sus $\Pr\{A|B\}$, obţinem

$$\Pr\{A|B\} = \frac{\Pr\{A\} \Pr\{B|A\}}{\Pr\{B\}}, \tag{6.21}$$

relaţie care este cunoscută sub numele de ***teorema lui Bayes***. Numărătorul $\Pr\{B\}$ este o constantă de normalizare pe care o putem rescrie după cum urmează. Deoarece $B = (B \cap A) \cup (B \cap \bar{A})$ şi $B \cap A$ şi $B \cap \bar{A}$ sunt mutual exclusive,

$$\Pr\{B\} = \Pr\{B \cap A\} + \Pr\{B \cap \bar{A}\} = \Pr\{A\}\Pr\{B|A\} + \Pr\{\bar{A}\}\Pr\{B|\bar{A}\}.$$

Înlocuind în ecuaţia (6.21), obţinem o formă echivalentă a teoremei lui Bayes:

$$\Pr\{A|B\} = \frac{\Pr\{A\}\Pr\{B|A\}}{\Pr\{A\}\Pr\{B|A\} + \Pr\{\bar{A}\}\Pr\{B|\bar{A}\}}.$$

Teorema lui Bayes poate uşura calculul probabilităţilor condiţionate. De exemplu, să presupunem că avem o monedă perfectă şi o monedă falsă care cade întotdeauna pe stemă. Efectuăm un experiment ce constă din trei evenimente independente: una dintre cele două monede este aleasă la întâmplare, aruncată o dată şi apoi aruncată din nou. Să presupunem că moneda aleasă a dat stema la ambele aruncări. Care este probabilitatea ca ea să fie falsă?

Vom rezolva această problemă utilizând teorema lui Bayes. Fie A evenimentul ca să fie aleasă moneda falsă şi fie B evenimentul ca moneda să ne dea de două ori stema. Dorim să determinăm $\Pr\{A|B\}$. Avem $\Pr\{A\} = 1/2$, $\Pr\{B|A\} = 1$, $\Pr\{\bar{A}\} = 1/2$ şi $\Pr\{B|\bar{A}\} = 1/4$; deci

$$\Pr\{A|B\} = \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)} = 4/5.$$

Exerciţii

6.2-1 Demonstraţi ***inegalitatea lui Boole***: pentru orice secvenţă finită sau infinit numărabilă de evenimente $A_1, A_2, \ldots$

$$\Pr\{A_1 \cup A_2 \cup \ldots\} \leq \Pr\{A_1\} + \Pr\{A_2\} + \cdots. \tag{6.22}$$

6.2-2 Profesorul Rosencrantz aruncă o monedă perfectă. Profesorul Guildenstern aruncă două monede perfecte. Care este probabilitatea ca profesorul Rosencrantz să obţină mai multe steme decât profesorul Guildenstern?

6.2-3 Un pachet de 10 cărţi, fiecare conţinând un număr distinct de la 1 la 10, este amestecat bine. Se extrag trei cărţi din pachet, câte una la un moment dat. Care este probabilitatea ca cele trei cărţi selectate să fie sortate (crescător)?

6.2-4 $\star$ Se dă o monedă falsă, care, atunci când este aruncată, produce stema cu o probabilitate p (necunoscută), unde $0 < p < 1$. Arătaţi cum poate fi simulată "o aruncare de monedă perfectă" examinând aruncări multiple. (*Indicaţie:* aruncaţi moneda de două ori şi apoi fie returnaţi rezultatul, fie repetaţi experimentul.) Demonstraţi că răspunsul este corect.

6.2-5 $\star$ Descrieţi o procedură are acceptă, la intrare, doi întregi a şi b, astfel încât $0 < a < b$ şi, utilizând aruncări de monede perfecte produceţi, la ieşire, stema cu probabilitatea a/b şi banul cu probabilitatea $(b-a)/b$. Daţi o margine a numărului mediu de aruncări de monede, care poate fi un polinom în $\lg b$.

6.2-6 Demonstraţi că

$$\Pr\{A|B\} + \Pr\{\bar{A}|B\} = 1.$$

6.2-7 Arătaţi că, pentru orice colecţie de evenimente $A_1, A_2, \ldots, A_n$,

$$\Pr\{A_1 \cap A_2 \cap \ldots \cap A_n\} = \\ \Pr\{A_1\} \cdot \Pr\{A_2|A_1\} \cdot \Pr\{A_3|A_1 \cap A_2\} \cdots \Pr\{A_n|A_1 \cap A_2 \cap \ldots \cap A_{n-1}\}.$$

6.2-8 ⋆ Arătaţi cum se poate construi o mulţime de n evenimente care să fie două câte două independente, dar orice submulţime de $k > 2$ dintre ele *să nu fie* mulţime de evenimente mutual independente.

6.2-9 ⋆ Două evenimente A şi B sunt ***independente condiţionat*** dacă

$$\Pr\{A \cap B|C\} = \Pr\{A|C\} \cdot \Pr\{B|C\},$$

unde C este dat. Daţi un exemplu simplu, dar netrivial, de două evenimente care nu sunt independente, dar care sunt independente condiţionat, fiind dat un al treilea eveniment.

6.2-10 ⋆ Sunteţi concurent într-un joc-spectacol în care un premiu este ascuns în spatele uneia dintre trei cortine. Veţi câştiga premiul dacă alegeţi cortina corectă. După ce aţi ales cortina, dar înainte ca ea să fie ridicată, prezentatorul ridică una dintre celelalte cortine, arătând o scenă goală şi vă întreabă dacă doriţi să vă schimbaţi opţiunea curentă pentru cortina rămasă. Care va fi şansa dumneavoastră dacă vă schimbaţi opţiunea?

6.2-11 ⋆ Un director al unei închisori a ales aleator un prizonier din trei pentru a fi eliberat. Ceilalţi doi vor fi executaţi. Gardianul ştie care va fi liber, dar îi este interzis să dea oricărui prizonier informaţii referitoare la starea acestuia. Să numim cei trei prizonieri X, Y şi Z. Prizonierul X întreabă gardianul, în particular, care dintre Y şi Z vor fi executaţi, argumentând că, dacă cel puţin unul dintre cei doi trebuie să moară, gardianul nu-i dă nici o informaţie despre starea sa. Gardianul îi spune lui X că Y va fi executat. Prizonierul X se simte mai fericit acum, deoarece sau el sau Z va fi liber, ceea ce înseamnă că probabilitatea ca el să fie liber este acum $1/2$. Are dreptate, sau şansa sa este tot $1/3$? Explicaţi.

6.3. Variabile aleatoare discrete

O ***variabilă aleatoare*** (***discretă***) X este o funcţie de la un spaţiu de selecţie S, finit sau infinit numărabil, la mulţimea numerelor reale. Ea asociază un număr real fiecărui rezultat al unui experiment, ceea ce ne permite să lucrăm cu distribuţia de probabilitate indusă de mulţimea de numere rezultată. Variabilele aleatoare pot fi definite şi pentru spaţii de selecţie nenumărabile, dar se ajunge la probleme tehnice a căror rezolvare nu este necesară pentru scopurile noastre. De aceea, vom presupune că variabilele aleatoare sunt discrete.

Pentru o variabilă aleatoare X şi un număr real x, definim evenimentul $X = x$ ca fiind $\{s \in S : X(s) = x\}$; astfel

$$\Pr\{X = x\} = \sum_{\{s \in S : X(s) = x\}} \Pr\{s\}.$$

Funcţia

$$f(x) = \Pr\{X = x\}$$

este ***densitatea de probabilitate***[5] a variabilei aleatoare X. Din axiomele probabilităţii, $\Pr\{X = x\} \geq 0$ şi $\sum_x \Pr\{X = x\} = 1$.

De exemplu, să considerăm experimentul aruncării unei perechi de zaruri obişnuite, cu 6 feţe. Există 36 de evenimente elementare posibile în spaţiul de selecţie. Presupunem că distribuţia de probabilitate este uniformă, astfel că fiecare eveniment elementar $s \in S$ are aceeaşi probabilitate: $\Pr\{s\} = 1/36$. Definim variabila aleatoare X ca fiind *maximul* celor două valori de pe feţele zarurilor. Avem $\Pr\{X = 3\} = 5/36$, deoarece X atribuie valoarea 3 la 5 din cele 36 de evenimente elementare posibile şi anume (1,3), (2,3), (3,3), (3,2) şi (3,1).

Se obişnuieşte ca mai multe variabile aleatoare să fie definite pe acelaşi spaţiu de selecţie. Dacă X şi Y sunt variabile aleatoare, funcţia

$$f(x, y) = \Pr\{X = x \text{ şi } Y = y\}$$

este ***funcţia densitate de probabilitate asociată*** lui X şi Y. Pentru o valoare fixată y,

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ şi } Y = y\}$$

şi la fel, pentru o valoare fixată x

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ şi } Y = y\}.$$

Utilizând definiţia (6.19) a probabilităţii condiţionate, avem

$$\Pr\{X = x | Y = y\} = \frac{\Pr\{X = x \text{ şi } Y = y\}}{\Pr\{Y = y\}}.$$

Spunem că două variabile aleatoare X şi Y sunt ***independente*** dacă, pentru orice x şi y, evenimentele $X = x$ şi $Y = y$ sunt independente, sau echivalent, dacă, pentru orice x şi y, avem $\Pr\{X = x \text{ şi } Y = y\} = \Pr\{X = x\} \Pr\{Y = y\}$.

Dându-se o mulţime de variabile aleatoare peste acelaşi spaţiu de selecţie, se pot defini noi variabile aleatoare, cum ar fi suma, produsul sau alte funcţii de variabilele originale.

Valoarea medie a unei variabile aleatoare

Cea mai simplă şi mai utilă descriere a distribuţiei unei variabile aleatoare este "media" valorilor pe care le ia. ***Valoarea medie*** (sau sinonim ***speranţa*** sau ***media***) a unei variabile aleatoare discrete X este

$$\mathrm{E}[X] = \sum_x x \Pr\{X = x\}, \tag{6.23}$$

care este bine definită dacă suma este finită sau absolut convergentă. Uneori media se notează cu μ_X, sau când variabila aleatoare este subînţeleasă din context cu μ.

Să considerăm un joc în care se aruncă două monede perfecte. Câştigaţi 3\$ pentru fiecare stemă şi pierdeţi 2\$ pentru fiecare ban. Valoarea medie a variabilei aleatoare X ce reprezintă câştigul dumneavoastră este

$$\mathrm{E}[X] = 6 \cdot \Pr\{2\text{S}\} + 1 \cdot \Pr\{1\text{S}, 1\text{B}\} - 4 \cdot \Pr\{2\text{B}\} = 6(1/4) + 1(1/2) - 4(1/4) = 1.$$

[5] termenul corect din punctul de vedere al teoriei probabilităţilor este ***funcţie de masă*** sau ***funcţie de frecvenţă*** – n.t.

Media sumei a două variabile aleatoare este suma mediilor, adică

$$\mathrm{E}[X+Y] = \mathrm{E}[X] + \mathrm{E}[Y], \tag{6.24}$$

ori de câte ori $\mathrm{E}[X]$ şi $\mathrm{E}[Y]$ sunt definite. Această proprietate se extinde la sume finite şi absolut convergente de medii.

Dacă X este o variabilă aleatoare, orice funcţie $g(x)$ defineşte o nouă variabilă aleatoare $g(X)$. Dacă valoarea medie a lui $g(X)$ este definită, atunci

$$\mathrm{E}[g(X)] = \sum_x g(x) \Pr\{X = x\}.$$

Luând $g(x) = ax$, avem pentru orice constantă

$$\mathrm{E}[aX] = a\mathrm{E}[X]. \tag{6.25}$$

În consecinţă, mediile sunt liniare: pentru oricare două variabile aleatoare X şi Y şi orice constantă a,

$$\mathrm{E}[aX+Y] = a\mathrm{E}[X] + \mathrm{E}[Y]. \tag{6.26}$$

Când două variabile aleatoare X şi Y sunt independente şi media fiecăreia este definită, avem

$$\begin{aligned}\mathrm{E}[XY] &= \sum_x \sum_y xy \Pr\{X = x \text{ şi } Y = y\} = \sum_x \sum_y xy \Pr\{X = x\} \Pr\{Y = y\} = \\ &= \left(\sum_x x \Pr\{X = x\}\right)\left(\sum_y y \Pr\{Y = y\}\right) = \mathrm{E}[X]\mathrm{E}[Y].\end{aligned}$$

În general, când n variabile aleatoare $X_1, X_2, \ldots, X_n$ sunt mutual independente

$$\mathrm{E}[X_1 X_2 \ldots X_n] = \mathrm{E}[X_1]\mathrm{E}[X_2]\ldots\mathrm{E}[X_n]. \tag{6.27}$$

Atunci când o variabilă aleatoare X ia valori din mulţimea numerelor naturale $\mathbb{N} = \{0, 1, 2, \ldots\}$, există o formulă convenabilă pentru media sa

$$\mathrm{E}[X] = \sum_{i=0}^{\infty} i \Pr\{X = i\} = \sum_{i=0}^{\infty} i \left(\Pr\{X \geq i\} - \Pr\{X \geq i+1\}\right) = \sum_{i=1}^{\infty} \Pr\{X \geq i\}, \tag{6.28}$$

deoarece fiecare termen $\Pr\{X \geq i\}$ este adunat de i ori şi scăzut de $i-1$ ori (exceptând $\Pr\{X \geq 0\}$ care este adunat de 0 ori şi nu este scăzut deloc).

Dispersie şi abatere medie pătratică

Dispersia unei variabile aleatoare X cu media $\mathrm{E}[X]$ este

$$\begin{aligned}\mathrm{Var}[X] &= E\left[(X - \mathrm{E}[X])^2\right] = E\left[X^2 - 2X\mathrm{E}[X] + E^2[X]\right] = \\ &= \mathrm{E}[X^2] - 2\mathrm{E}[X\mathrm{E}[X]] + E^2[X] = \mathrm{E}[X^2] - 2E^2[X] + E^2[X] = \\ &= \mathrm{E}[X^2] - E^2[X]. \end{aligned} \tag{6.29}$$

Justificarea pentru egalităţile $\mathrm{E}[E^2[X]] = E^2[X]$ şi $\mathrm{E}[X\mathrm{E}[X]] = E^2[X]$ este aceea că $\mathrm{E}[X]$ nu este o variabilă aleatoare, ci pur şi simplu un număr real, ceea ce înseamnă că se aplică ecuaţia (6.25) (cu $a = \mathrm{E}[X]$). Ecuaţia (6.29) poate fi rescrisă pentru a obţine o expresie a mediei pătratului unei variabile aleatoare:

$$\mathrm{E}[X^2] = \mathrm{Var}[X] + E^2[X]. \tag{6.30}$$

Dispersia unei variabile aleatoare X şi dispersia lui aX sunt legate prin

$$\mathrm{Var}[aX] = a^2\mathrm{Var}[X].$$

Când X şi Y sunt variabile aleatoare independente,

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y].$$

În general, dacă n variabile aleatoare $X_1, X_2, \ldots, X_n$ sunt două câte două independente, atunci

$$Var\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i]. \tag{6.31}$$

Abaterea medie pătratică (***deviaţia standard***) a variabilei aleatoare X este rădăcina pătrată pozitivă a dispersiei lui X. Abaterea medie pătratică a unei variabile aleatoare X se notează uneori cu σ_X sau cu σ, când variabila aleatoare este subînţeleasă din context. Cu această notaţie, dispersia lui X se notează cu σ^2.

Exerciţii

6.3-1 Se aruncă două zaruri obişnuite cu 6 feţe. Care este media sumei celor două valori arătate de zaruri? Care este media maximului celor două valori arătate?

6.3-2 Un tablou $A[1..n]$ conţine n numere distincte ordonate aleator, fiecare permutare a celor n numere având aceeaşi probabilitate de apariţie. Care este media indicelui elementului maxim al tabloului? Care este media indicelui elementului minim al tabloului?

6.3-3 La un joc de carnaval se pun trei zaruri într-o ceaşcă. Un jucător poate paria 1\$ pe orice număr de la 1 la 6. Ceaşca este scuturată, zarurile sunt aruncate, iar plata se face în modul următor. Dacă numărul jucătorului nu apare pe nici unul din zaruri, el îşi pierde dolarul. Altfel, dacă numărul său apare pe exact k din cele trei zaruri, $k = 1, 2, 3$, el îşi păstrează dolarul şi mai câştigă k dolari. Care este câştigul mediu când jocul se joacă o singură dată?

6.3-4 $\star$ Fie X şi Y variabile aleatoare independente. Arătaţi că $f(X)$ şi $g(X)$ sunt independente pentru orice alegere a funcţiilor f şi g.

6.3-5 $\star$ Fie X o variabilă aleatoare nenegativă şi presupunem că $\mathrm{E}[X]$ este bine definită. Demonstraţi ***inegalitatea lui Markov***:

$$\Pr\{X \geq t\} \leq \mathrm{E}[X]/t \tag{6.32}$$

pentru orice $t > 0$.

6.3-6 ⋆ Fie S un spaţiu de selecţie şi X şi X' variabile aleatoare astfel încât $X(s) \geq X'(s)$ pentru orice $s \in S$. Demonstraţi că pentru orice constantă reală t,

$$\Pr\{X \geq t\} \geq \Pr\{X' \geq t\}.$$

6.3-7 Ce este mai mare: media pătratului unei variabile aleatoare sau pătratul mediei?

6.3-8 Arătaţi că, pentru orice variabilă aleatoare X ce ia numai valorile 0 şi 1, avem $\mathrm{Var}[X] = \mathrm{E}[X]\mathrm{E}[1 - X]$.

6.3-9 Demonstraţi că $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$ folosind definiţia (6.29) a dispersiei.

6.4. Distribuţia geometrică şi distribuţia binomială

Aruncarea unei monede este un exemplu de ***probă bernoullianǎ***, care este definită ca un experiment cu numai două rezultate posibile: ***succes***, care apare cu probabilitatea p şi ***eşec***, care apare cu probabilitatea $q = 1 - p$. Când vorbim de ***probe bernoulliene*** în mod colectiv, înseamnă că probele sunt mutual independente şi, dacă nu se specifică altceva, că fiecare are aceeaşi probabilitate de succes p. Două distribuţii importante derivă din probele bernoulliene: distribuţia geometrică şi cea binomială.

Distribuţia geometrică

Presupunem că avem o secvenţă de probe bernoulliene, fiecare cu probabilitatea de succes p şi probabilitatea de eşec $q = 1 - p$. Câte încercări apar înainte de a obţine un succes? Fie variabila aleatoare X numărul de încercări necesare pentru a obţine un succes. Atunci X ia valori în domeniul $\{1, 2, \ldots\}$ şi pentru $k \geq 1$

$$\Pr\{X = k\} = q^{k-1}p, \tag{6.33}$$

deoarece avem $k - 1$ eşecuri înaintea unui succes. O distribuţie de probabilitate ce satisface ecuaţia (6.33) se numeşte ***distribuţie geometrică***. Figura 6.1 ilustrează o astfel de distribuţie.

Presupunând că $p < 1$, media distribuţiei geometrice poate fi calculată utilizând identitatea (3.6):

$$\mathrm{E}[X] = \sum_{k=1}^{\infty} kq^{k-1}p = \frac{p}{q}\sum_{k=0}^{\infty} kq^k = \frac{p}{q} \cdot \frac{q}{(1-q)^2} = 1/p. \tag{6.34}$$

Astfel, în medie, este nevoie de $1/p$ încercări pentru a obţine un succes, rezultat intuitiv. Dispersia, care se poate calcula în acelaşi mod, este

$$\mathrm{Var}[X] = q/p^2. \tag{6.35}$$

De exemplu, să presupunem că aruncăm, repetat, două zaruri până când obţinem fie suma şapte, fie suma unsprezece. Din cele 36 de rezultate posibile, 6 ne dau suma şapte şi 2 suma unsprezece. Astfel probabilitatea de succes este $p = 8/36 = 2/9$ şi trebuie să aruncăm zarurile, în medie, de $1/p = 9/2 = 4.5$ ori pentru a obţine suma şapte sau unsprezece.

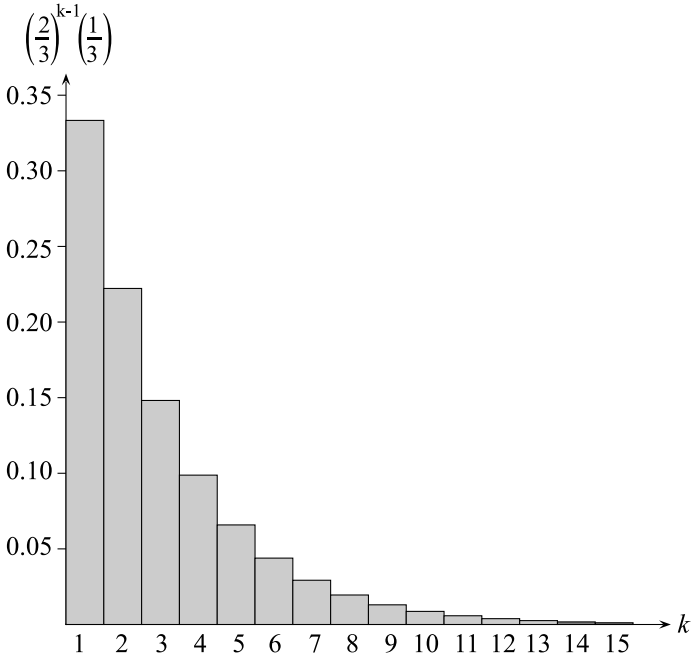


Figura 6.1 O distribuţie geometrică cu probabilitatea de succes $p = 1/3$ şi probabilitatea de eşec $q = 1 - p$. Media distribuţiei este $1/p = 3$.

Distribuţia binomială

Câte succese apar în n probe bernouliene, unde un succes apare cu probabilitatea p şi un eşec cu probabilitatea $q = 1 - p$? Definim variabila aleatoare X ca fiind numărul de succese în n probe. Atunci X ia valori în domeniul $\{0, 1, \ldots, n\}$ şi pentru $k = 0, \ldots, n$,

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}, \tag{6.36}$$

deoarece sunt $\binom{n}{k}$ moduri de a alege care k probe din totalul celor n sunt succese şi probabilitatea de apariţie a fiecăreia dintre ele este $p^k q^{n-k}$. O distribuţie de probabilitate ce satisface ecuaţia (6.36) se numeşte ***distribuţie binomială***. Pentru comoditate, definim familia distribuţiilor binomiale utilizând notaţia

$$b(k; n, p) = \binom{n}{k} p^k (1 - p)^{n-k}. \tag{6.37}$$

Figura 6.2 ilustrează o distribuţie binomială. Numele de "binomial" vine de la faptul că (6.37) este al k-lea termen al dezvoltării lui $(p + q)^n$. În consecinţă, deoarece $p + q = 1$,

$$\sum_{k=0}^{n} b(k; n, p) = 1, \tag{6.38}$$

aşa cum cere axioma 2 de probabilitate.

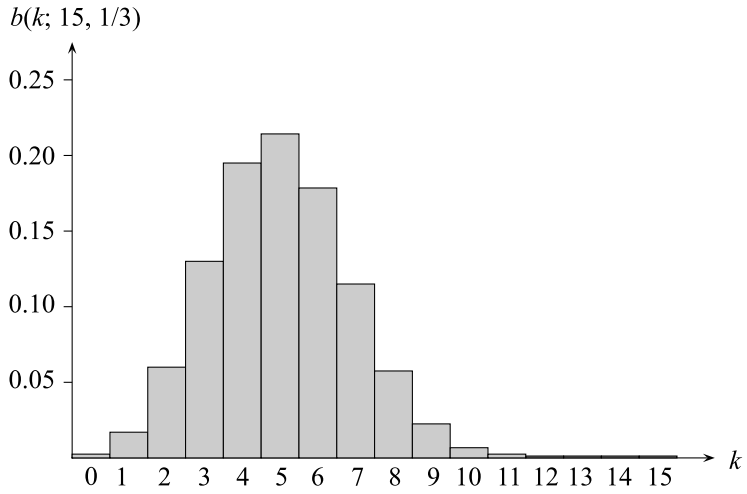


Figura 6.2 Distribuţia binomială $b(k; 15, 1/3)$ ce rezultă din $n = 15$ probe bernoulliene, fiecare cu probabilitatea de succes $p = 1/3$. Media distribuţiei este $np = 5$.

Putem calcula media unei variabile aleatoare având o distribuţie binomială din ecuaţiile (6.14) şi (6.38). Fie X o variabilă aleatoare ce urmează distribuţia binomială $b(k; n, p)$ şi fie $q = 1 - p$. Din definiţia mediei avem

$$\begin{aligned} \mathrm{E}[X] &= \sum_{k=0}^{n} k b(k; n, p) = \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} = np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} = \\ &= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} = np \sum_{k=0}^{n-1} b(k; n-1, p) = np. \end{aligned} \tag{6.39}$$

Utilizând liniaritatea mediei, putem obţine acelaşi rezultat cu substanţial mai puţină algebră. Fie X_i o variabilă aleatoare ce descrie numărul de succese la a i-a probă. Atunci, $\mathrm{E}[X_i] = p \cdot 1 + q \cdot 0 = p$ şi, din liniaritatea mediei (6.26), numărul mediu de succese din n încercări este

$$\mathrm{E}[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} p = np.$$

Aceeaşi abordare poate fi folosită la calculul dispersiei. Utilizând ecuaţia (6.29), avem $\mathrm{Var}[X_i] = \mathrm{E}[X_i^2] - E^2[X_i]$. Deoarece X_i ia numai valorile 0 şi 1, avem $\mathrm{E}[X_i^2] = \mathrm{E}[X_i] = p$ şi deci

$$\mathrm{Var}[X_i] = p - p^2 = pq. \tag{6.40}$$

Pentru a calcula dispersia lui X, vom exploata avantajul independenţei celor n încercări; astfel din ecuaţia (6.31)

$$\mathrm{Var}[X] = \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i] = \sum_{i=1}^{n} pq = npq. \tag{6.41}$$

Conform figurii 6.2, distribuţia binomială $b(k;n,p)$ creşte pe măsură ce k variază de la 0 la n, până când atinge media np şi apoi scade. Putem demonstra că distribuţia se comportă întotdeauna în acest mod, examinând raportul termenilor succesivi

$$\begin{aligned}\frac{b(k;n,p)}{b(k-1;n,p)} &= \frac{\binom{n}{k}p^k q^{n-k}}{\binom{n}{k-1}p^{k-1}q^{n-k+1}} = \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} = \\ &= \frac{(n-k+1)p}{kq} = 1 + \frac{(n+1)p-k}{kq}. \end{aligned} \tag{6.42}$$

Acest raport este mai mare decât 1 când $(n+1)p-k$ este pozitiv. În consecinţă, $b(k;n,p) > b(k-1;n,p)$ pentru $k < (n+1)p$ (distribuţia creşte) şi $b(k;n,p) < b(k-1;n,p)$ pentru $k > (n+1)p$ (distribuţia descreşte). Dacă $k = (n+1)p$ este întreg, atunci $b(k;n,p) = b(k-1;n,p)$ şi distribuţia are două maxime: în $k = (n+1)p$ şi în $k-1 = (n+1)p-1 = np-q$. Altfel, distribuţia îşi atinge maximul în întregul k unic, situat în domeniul $np-q < k < (n+1)p$.

Lema următoare ne dă o margine superioară a distribuţiei binomiale.

Lema 6.1 Fie $n \geq 0$, $0 < p < 1$, $q = 1-p$ şi $0 \leq k \leq n$. Atunci

$$b(k;n,p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

Demonstraţie. Utilizând ecuaţia (6.10) avem

$$b(k;n,p) = \binom{n}{k}p^k q^{n-k} \leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} = \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

■

Exerciţii

6.4-1 Verificaţi a doua axiomă a probabilităţii pentru distribuţia geometrică.

6.4-2 De câte ori trebuie aruncate, în medie, 6 monede perfecte înainte de a obţine 3 steme şi 3 bani?

6.4-3 Arătaţi că $b(k;n,p) = b(n-k;n,q)$, unde $q = 1-p$.

6.4-4 Arătaţi că valoarea maximă a distribuţiei binomiale $b(k;n,p)$ este aproximativ $1/\sqrt{2\pi npq}$, unde $q = 1-p$.

6.4-5 ⋆ Arătaţi că probabilitatea de a nu avea nici un succes din n probe bernoulliene, fiecare cu probabilitatea $p = 1/n$, este aproximativ $1/e$. Arătaţi că probabilitatea de a avea exact un succes este, de asemenea, aproximativ $1/e$.

6.4-6 ⋆ Profesorul Rosencrantz aruncă o monedă perfectă de n ori şi la fel face şi profesorul Guildenstern. Arătaţi că probabilitatea ca ei să obţină acelaşi număr de steme este egală cu $\binom{2n}{n}/4^n$. (*Indicaţie:* pentru profesorul Rosencrantz considerăm stema succes; pentru profesorul Guildenstern considerăm banul succes). Utilizaţi raţionamentul pentru a verifica identitatea

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}.$$

6.4-7 ⋆ Arătaţi că pentru $0 \le k \le n$,

$$b(k; n, 1/2) \le 2^{nH(k/n)-n},$$

unde $H(x)$ este funcţia entropie binară (6.13).

6.4-8 ⋆ Considerăm n probe bernoulliene, unde, pentru $i = 1, 2, \ldots, n$, a i-a probă are probabilitatea de succes p_i şi fie X variabila aleatoare ce desemnează numărul total de succese. Fie $p \ge p_i$ pentru orice $i = 1, 2, \ldots, n$. Demonstraţi că, pentru $1 \le k \le n$,

$$\Pr\{X < k\} \le \sum_{i=0}^{k-1} b(i; n, p).$$

6.4-9 Fie X o variabilă aleatoare ce reprezintă numărul total de succese într-o mulţime A de n probe bernoulliene, unde a i-a probă are probabilitatea de succes p_i, şi fie X' variabila aleatoare ce reprezintă numărul total de succese dintr-o mulţime A' de n probe bernoulliene, unde a i-a probă are o probabilitate de succes $p'_i \ge p_i$. Demonstraţi că, pentru $0 \le k \le n$,

$$\Pr\{X' \ge k\} \ge \Pr\{X \ge k\}.$$

(*Indicaţie:* Arătaţi cum se pot obţine probele bernoulliene din A' printr-un experiment ce implică probele din A şi utilizaţi rezultatul exerciţiului 6.3-6.)

6.5. Cozile distribuţiei binomiale

Probabilitatea de a avea cel puţin sau cel mult k succese în n probe bernoulliene, fiecare cu probabilitatea de succes p, este, adesea, de mai mare interes decât probabilitatea de a avea exact k succese. În această secţiune vom investiga ***cozile*** distribuţiei binomiale, adică cele două regiuni ale distribuţiei $b(k; n, p)$ care sunt îndepărtate de media np.[6] Vom demonstra câteva margini importante (ale sumei tuturor termenilor) dintr-o coadă. Vom demonstra, mai întâi, o margine pentru coada dreaptă a distribuţiei $b(k; n, p)$. Marginile pentru coada stângă pot fi determinate din cele pentru coada dreaptă schimbând între ele rolurile succeselor şi eşecurilor.

Teorema 6.2 Considerăm o secvenţă de n probe bernoulliene, unde succesul apare cu probabilitatea p. Fie X o variabilă aleatoare ce desemnează numărul total de succese. Atunci pentru $0 \le k \le n$, probabilitatea să avem cel puţin k succese este

$$\Pr\{X \ge k\} = \sum_{i=k}^{n} b(i; n, p) \le \binom{n}{k} p^k.$$

Demonstraţie. Vom utiliza inegalitatea (6.15)

$$\binom{n}{k+i} \le \binom{n}{k}\binom{n-k}{i}.$$

[6] Regiuni de forma $\{X \ge k\}$ sau $\{X \le k\}$ – n.t.

Avem

$$\begin{aligned}\Pr\{X \geq k\} &= \sum_{i=k}^{n} b(i;n,p) = \sum_{i=0}^{n-k} b(k+i;n,p) = \sum_{i=0}^{n-k} \binom{n}{k+i} p^{k+i}(1-p)^{n-(k+i)} \\ &\leq \sum_{i=0}^{n-k} \binom{n}{k}\binom{n-k}{i} p^{k+i}(1-p)^{n-(k+i)} = \binom{n}{k} p^k \sum_{i=0}^{n-k} \binom{n-k}{i} p^i (1-p)^{(n-k)-i} \\ &= \binom{n}{k} p^k \sum_{i=0}^{n-k} b(i;n-k,p) = \binom{n}{k} p^k,\end{aligned}$$

căci $\sum_{i=0}^{n-k} b(i;n-k,p) = 1$, din ecuaţia (6.38). ∎

Corolarul următor reformulează teorema pentru coada stângă a distribuţiei binomiale. În general, vom lăsa în seama cititorului trecerea de la marginea unei cozi la alta.

Corolarul 6.3 Considerăm o secvenţă de n probe bernoulliene, unde succesul apare cu probabilitatea p. Dacă X este o variabilă aleatoare ce reprezintă numărul total de succese, atunci, pentru $0 \leq k \leq n$, probabilitatea să avem cel mult k succese este

$$\Pr\{X \leq k\} = \sum_{i=0}^{k} b(i;n,p) \leq \binom{n}{n-k}(1-p)^{n-k} = \binom{n}{k}(1-p)^{n-k}.$$

∎

Marginea următoare se concentrează asupra cozii stângi a distribuţiei binomiale. Cu cât ne îndepărtăm de medie, numărul de succese din coada stângă scade exponenţial, aşa cum ne arată teorema următoare.

Teorema 6.4 Considerăm o secvenţă de n probe bernoulliene, în care succesul apare cu probabilitatea p şi eşecul cu probabilitatea $q = 1-p$. Fie X o variabilă aleatoare ce reprezintă numărul de succese. Atunci, pentru $0 < k < np$, probabilitatea a mai puţin de k succese este

$$\Pr\{X < k\} = \sum_{i=0}^{k-1} b(i;n,p) < \frac{kq}{np-k} b(k;n,p).$$

Demonstraţie. Vom majora seria $\sum_{i=0}^{k-1} b(i;n,p)$ cu o serie geometrică utilizând tehnica din secţiunea 3.2. Pentru $i = 1, 2, \ldots, k$ din ecuaţia (6.42) avem

$$\frac{b(i-1;n,p)}{b(i;n,p)} = \frac{iq}{(n-i+1)p} < \left(\frac{i}{n-i}\right)\left(\frac{q}{p}\right) \leq \left(\frac{k}{n-k}\right)\left(\frac{q}{p}\right).$$

Dacă luăm

$$x = \left(\frac{k}{n-k}\right)\left(\frac{q}{p}\right) < 1,$$

rezultă că

$$b(i-1;n,p) < x\,b(i;n,p),$$

pentru $0 < i \leq k$. Iterând, obţinem

$$b(i; n, p) < x^{k-i} b(k; n, p)$$

pentru $0 \leq i < k$ şi deci

$$\sum_{i=0}^{k-1} b(i; n, p) < \sum_{i=0}^{k-1} x^{k-i} b(k; n, p) < b(k; n, p) \sum_{i=1}^{\infty} x^i = \frac{x}{1-x} b(k; n, p) = \frac{kq}{np-k} b(k; n, p).$$

■

Când $k \leq np/2$, avem $kq/(np-k) \leq 1$, ceea ce înseamnă că $b(k; n, p)$ majorează suma tuturor termenilor mai mici decât k. De exemplu, să presupunem că aruncăm n monede perfecte. Teorema 6.4 ne spune că, luând $p = 1/2$ şi $k = n/4$, probabilitatea de a obţine mai puţin de $n/4$ steme este mai mică decât probabilitatea de a obţine $n/4$ steme. Mai mult, pentru orice $r \geq 4$, probabilitatea de a obţine mai puţin de n/r steme este mai mică decât probabilitatea de a obţine exact n/r steme. Teorema 6.4 poate fi, de asemenea, utilă în combinaţie cu margini superioare ale distribuţiei binomiale cum ar fi lema 6.1. O margine a cozii drepte se poate determina la fel.

Corolarul 6.5 Considerăm o secvenţă de n probe bernoulliene, în care succesul apare cu probabilitatea p. Fie X o variabilă aleatoare ce reprezintă numărul total de succese. Atunci, pentru $np < k < n$, probabilitatea de a avea mai mult de k succese este

$$\Pr\{X > k\} = \sum_{i=k+1}^{n} b(i; n, p) < \frac{(n-k)p}{k-np} b(k; n, p).$$

■

Teorema următoare consideră n probe bernoulliene, fiecare cu o probabilitate p_i de succes, pentru $i = 1, 2, \ldots, n$. Aşa cum ne arată corolarul următor, putem utiliza teorema pentru a furniza o margine superioară a cozii drepte a distribuţiei binomiale, punând la fiecare încercare $p = p_i$.

Teorema 6.6 Considerăm o secvenţă de n probe bernoulliene, unde, la a i-a probă, pentru $i = 1, 2, \ldots, n$, succesul apare cu probabilitatea p_i, iar eşecul cu probabilitatea $q_i = 1 - p_i$. Fie X o variabilă aleatoare ce reprezintă numărul total de succese şi fie $\mu = E[X]$. Atunci, pentru $r > \mu$,

$$\Pr\{X - \mu \geq r\} \leq \left(\frac{\mu e}{r}\right)^r.$$

Demonstraţie. Deoarece, pentru orice $\alpha > 0$, funcţia $e^{\alpha x}$ este strict crescătoare în x, avem

$$\Pr\{X - \mu \geq r\} = \Pr\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\},$$

unde α va fi determinat mai târziu. Utilizând inegalitatea lui Markov (6.32) obţinem

$$\Pr\{X - \mu \geq r\} \leq E\left[e^{\alpha(X-\mu)}\right] e^{-\alpha r}. \tag{6.43}$$

Partea cea mai consistentă a demonstraţiei o constituie mărginirea lui $E\left[e^{\alpha(X-\mu)}\right]$ şi substituirea lui α în inegalitea (6.43) cu o valoare potrivită. Mai întâi evaluăm $E\left[e^{\alpha(X-\mu)}\right]$. Pentru $i =$

$1, 2, \ldots, n$, fie X_i o variabilă aleatoare care are valoarea 1, dacă a i-a probă bernoullianǎ este un succes şi 0, dacă este un eşec. Astfel

$$X = \sum_{i=1}^{n} X_i$$

şi

$$X - \mu = \sum_{i=1}^{n} (X_i - p_i).$$

Înlocuind pentru $X - \mu$, obţinem

$$E\left[e^{\alpha(X-\mu)}\right] = E\left[\prod_{i=1}^{n} e^{\alpha(X_i - p_i)}\right] = \prod_{i=1}^{n} E\left[e^{\alpha(X_i - p_i)}\right],$$

care rezultă din (6.27) deoarece independenţa mutuală a variabilelor aleatoare X_i implică independenţa mutuală a variabilelor aleatoare $e^{\alpha(X_i - p_i)}$ (vezi exerciţiul 6.3-4). Din definiţia mediei

$$\begin{aligned} E\left[e^{\alpha(X_i - p_i)}\right] &= e^{\alpha(1-p_i)} p_i + e^{\alpha(0-p_i)} q_i = p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq \\ &\leq p_i e^{\alpha} + 1 \leq \exp(p_i e^{\alpha}), \end{aligned} \tag{6.44}$$

unde $\exp(x)$ reprezintă funcţia exponenţială: $\exp(x) = e^x$. (Inegalitatea (6.44) rezultă din inegalităţile $\alpha > 0$, $q_i \leq 1$, $e^{\alpha q_i} \leq e^{\alpha}$ şi $e^{-\alpha p_i} \leq 1$, iar ultima linie rezultă din inegalitatea (2.7)). În consecinţă,

$$E\left[e^{\alpha(X-\mu)}\right] \leq \prod_{i=1}^{n} \exp(p_i e^{\alpha}) = \exp(\mu e^{\alpha})$$

căci $\mu = \sum_{i=1}^{n} p_i$. Deci, din inegalitatea (6.43), rezultă că

$$\Pr\{X - \mu \geq r\} \leq \exp(\mu e^{\alpha} - \alpha r). \tag{6.45}$$

Alegând $\alpha = \ln(r/\mu)$ (vezi exerciţiul 6.5-6), obţinem

$$\Pr\{X - \mu \geq r\} \leq \exp\left(\mu e^{\ln(r/\mu)} - r \ln(r/\mu)\right) = \exp(r - r \ln(r/\mu)) = \frac{e^r}{(r/\mu)^r} = \left(\frac{\mu e}{r}\right)^r.$$

■

Când se aplică probelor bernoulliene, în care la fiecare încercare avem aceeaşi probabilitate de succes, teorema 6.6 ne dă următorul corolar de mărginire a cozii drepte a unei distribuţii binomiale.

Corolarul 6.7 Considerăm o secvenţă de n probe bernoulliene, unde, la fiecare probă, succesul apare cu probabilitatea p, iar eşecul cu probabilitatea $q = 1 - p$. Atunci, pentru $r > np$,

$$\Pr\{X - np \geq r\} = \sum_{k=\lceil np+r \rceil}^{n} b(k; n, p) \leq \left(\frac{npe}{r}\right)^r.$$

Demonstraţie. Pentru o distribuţie binomială, (6.39) implică $\mu = E[X] = np$. ■

Exerciţii

6.5-1 ⋆ Ce este mai puţin probabil: să nu obţinem nici o stemă când aruncăm o monedă perfectă de n ori sau să obţinem mai puţin de n steme când aruncăm moneda de $4n$ ori?

6.5-2 ⋆ Arătaţi că

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na-k(a+1)} b(k; n, a/(a+1)),$$

pentru orice $a > 0$ şi orice k, astfel încât $0 < k < n$.

6.5-3 ⋆ Demonstraţi că, dacă $0 < k < np$, unde $0 < p < 1$ şi $q = 1 - p$, atunci

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np-k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}.$$

6.5-4 ⋆ Arătaţi că ipotezele teoremei 6.6 implică

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r.$$

La fel, arătaţi că ipotezele corolarului 6.7 implică

$$\Pr\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r.$$

6.5-5 ⋆ Considerăm o secvenţă de n probe bernoulliene, unde la a i-a probă, pentru $i = 1, 2, \ldots, n$ succesul apare cu probabilitatea p_i, iar eşecul cu probabilitatea $q_i = 1 - p_i$. Fie X o variabilă aleatoare ce reprezintă numărul total de succese şi fie $\mu = \mathrm{E}[X]$. Arătaţi că, pentru $r \geq 0$,

$$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n}.$$

(*Indicaţie:* Demonstraţi că $p_i e^{\alpha q_i} + q_i e^{1-\alpha p_i} \leq e^{-\alpha^2/2}$. Apoi urmaţi mersul demonstraţiei teoremei 6.6, utilizând această inegalitate în locul inegalităţii (6.44).)

6.5-6 ⋆ Arătaţi că, alegând $\alpha = \ln(r/\mu)$, se minimizează membrul drept al inegalităţii (6.45).

6.6. Analiză probabilistică

Această secţiune utilizează trei exemple pentru a ilustra analiza probabilistică. Primul determină probabilitatea ca, într-o cameră în care sunt k persoane, să existe o anumită pereche care să-şi sărbătorească ziua de naştere în aceeaşi zi. Al doilea exemplu ilustrează aruncarea aleatoare a bilelor în cutii. Al treilea investighează secvenţele de steme consecutive la aruncarea monedei.

6.6.1. Paradoxul zilei de naştere

Un exemplu bun pentru ilustrarea raţionamentului probabilist este clasicul ***paradox al zilei de naştere***. Câţi oameni trebuie să fie într-o încăpere pentru a avea o bună şansă să găsim doi născuţi în aceeaşi zi a anului? Răspunsul este că trebuie să fie surprinzător de puţini. Paradoxul este acela că, de fapt, trebuie mult mai puţini decât numărul de zile dintr-un an, aşa cum se va vedea. Pentru a răspunde la întrebare, să numerotăm oamenii din cameră cu întregii $1, 2, \ldots, k$, unde k este numărul de oameni din încăpere. Vom ignora anii bisecţi şi vom presupune că toţi anii au $n = 365$ de zile. Pentru $i = 1, 2, \ldots, k$, fie b_i ziua din an în care cade ziua de naştere a lui i, unde $1 \leq b_i \leq n$. Presupunem că zilele de naştere sunt uniform distribuite în cele n zile ale anului, aşa că $\Pr\{b_i = r\} = 1/n$, pentru $i = 1, 2, \ldots, k$ şi $r = 1, 2, \ldots, n$.

Probabilitatea ca două persoane i şi j să aibă aceeaşi zi de naştere depinde de faptul că selecţia aleatoare este dependentă sau independentă. Dacă zilele de naştere sunt independente, atunci probabilitatea ca ziua de naştere a lui i şi ziua de naştere a lui j să cadă amândouă în ziua r este

$$\Pr\{b_i = r \text{ şi } b_j = r\} = \Pr\{b_i = r\} \Pr\{b_j = r\} = 1/n^2.$$

Astfel, probabilitatea ca ambele să cadă în aceeaşi zi este

$$\Pr\{b_i = b_j\} = \sum_{r=1}^{n} \Pr\{b_i = r \text{ şi } b_j = r\} = \sum_{r=1}^{n} \frac{1}{n^2} = 1/n.$$

Intutiv, o dată ales b_i, probabilitatea ca b_j să fie ales acelaşi, este $1/n$. Astfel, probabilitatea ca i şi j să aibă aceeaşi zi de naştere este egală cu probabilitatea ca una dintre zilele de naştere să cadă într-o zi dată. De notat, totuşi, că această coincidenţă depinde de presupunerea că zilele de naştere sunt independente.

Putem analiza probabilitatea ca cel puţin 2 din k oameni să aibă aceeaşi zi de naştere, analizând evenimentul complementar. Probabilitatea ca cel puţin două zile de naştere să coincidă este 1 minus probabilitatea ca toate zilele de naştere să fie diferite. Evenimentul ca toate cele k persoane să aibă zile de naştere distincte este

$$B_k = \bigcap_{i=1}^{k-1} A_i,$$

unde A_i este evenimentul ca ziua de naştere a persoanei $(i+1)$ să fie diferită de ziua de naştere a persoanei j, pentru orice $j \leq i$, adică,

$$A_i = \{b_{i+1} \neq b_j : j = 1, 2, \ldots, i\}.$$

Deoarece putem scrie $B_k = A_{k-1} \cap B_{k-1}$, din ecuaţia (6.20) obţinem recurenţa

$$\Pr\{B_k\} = \Pr\{B_{k-1}\} \Pr\{A_{k-1} | B_{k-1}\}, \tag{6.46}$$

unde luăm $\Pr\{B_1\} = 1$ drept condiţie iniţială. Cu alte cuvinte, probabilitatea ca b_1, b_2, $\ldots$, b_k să fie zile de naştere distincte este probabilitatea ca b_1, b_2, $\ldots$, b_{k-1} să fie distincte înmulţită cu probabilitatea ca $b_k \neq b_i$, pentru $i = 1, 2, \ldots, k-1$, ştiind că $b_1, b_2, \ldots, b_{k-1}$ sunt distincte.

Dacă $b_1, b_2, \ldots, b_{k-1}$ sunt distincte, probabilitatea condiţionată ca $b_k \neq b_i$ pentru $i = 1, 2, \ldots, k-1$ este $(n-k+1)/n$, deoarece printre cele n zile există $n-(k-1)$ care nu sunt luate în considerare. Iterând recurenţa (6.46), obţinem

$$\begin{aligned}\Pr\{B_k\} &= \Pr\{B_1\}\Pr\{A_1|B_1\}\Pr\{A_2|B_2\}\ldots\Pr\{A_{k-1}|B_{k-1}\} = \\ &= 1\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\ldots\left(\frac{n-k+1}{n}\right) = \\ &= 1\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\ldots\left(1-\frac{k-1}{n}\right).\end{aligned}$$

Inegalitatea (2.7), $1+x \leq e^x$ ne dă

$$\Pr\{B_k\} \leq e^{-1/n}e^{-2/n}\ldots e^{-(k-1)n} = e^{-\sum_{i=1}^{k-1} i/n} = e^{-k(k-1)/2n} \leq 1/2$$

când $-k(k-1)/2n \leq \ln(1/2)$. Probabilitatea ca toate cele k zile de naştere să fie distincte este cel mult $1/2$ când $k(k-1) \geq 2n\ln 2$, adică pentru $k \geq (1+\sqrt{1+(8\ln 2)n})/2$, rezultat obţinut rezolvând inecuaţia pătratică. Pentru $n = 365$, trebuie să avem $k \geq 23$. Astfel, dacă în încăpere sunt cel puţin 23 de oameni, probabilitatea ca cel puţin doi să aibă aceeaşi zi de naştere este cel puţin $1/2$. Pe Marte, unde anul are 669 de zile, este nevoie de 31 de marţieni pentru a obţine aceeaşi probabilitate.

O altă metodă de analiză

Putem utiliza liniaritatea mediei (ecuaţia (6.26)) pentru o obţine o analiză mai simplă, dar aproximativă, a paradoxului zilei de naştere. Pentru fiecare pereche (i, j) dintre cele k persoane din încăpere, să definim variabilele aleatoare X_{ij}, pentru $1 \leq i < j \leq k$, prin

$$X_{ij} = \begin{cases} 1 & \text{dacă persoanele } i \text{ şi } j \text{ au aceeaşi zi de naştere} \\ 0 & \text{în caz contrar.} \end{cases}$$

Probabilitatea ca două persoane să aibă aceeaşi zi de naştere este $1/n$ şi astfel, conform definiţiei mediei (6.23),

$$E[X_{ij}] = 1\cdot(1/n) + 0\cdot(1-1/n) = 1/n.$$

Numărul mediu de perechi de indivizi având aceeaşi zi de naştere este, conform ecuaţiei (6.24), suma mediilor individuale ale tuturor perechilor, care este

$$\sum_{i=2}^{k}\sum_{j=1}^{i-1} E[X_{ij}] = \binom{k}{2}\frac{1}{n} = \frac{k(k-1)}{2n}.$$

Când $k(k-1) \geq 2n$, numărul mediu de zile de naştere este cel puţin 1. Astfel, dacă avem cel puţin $\sqrt{2n}$ persoane în încăpere, ne putem aştepta ca cel puţin două să aibă aceeaşi zi de naştere. Pentru $n = 365$, dacă $k = 28$, numărul mediu de persoane cu aceeaşi zi de naştere este $(28\cdot 27)/(2\cdot 365) \approx 1,0536$. Astfel, dacă avem cel puţin 28 de persoane, ne aşteptăm să găsim cel puţin o pereche având aceeaşi zi de naştere. Pe Marte, unde un an este de 669 de zile, avem nevoie de cel puţin 38 de marţieni.

Prima analiză a determinat numărul de persoane necesar pentru ca probabilitatea existenţei unei coincidenţe a zilelor de naştere să fie mai mare decât $1/2$, iar a două analiză a determinat

numărul de persoane necesar pentru ca numărul mediu de coincidenţe de zile de naştere să fie cel puţin 1. Deşi numărul de oameni determinat în fiecare din cele două situaţii diferă, ambele sunt asimptotic la fel: $\Theta(\sqrt{n})$.

6.6.2. Bile şi cutii

Considerăm procesul de aruncare aleatoare a unor bile identice în b cutii, numerotate cu $1, 2, \ldots, b$. Aruncările sunt independente şi, la fiecare aruncare, bila are aceeaşi probabilitate de a nimeri în oricare dintre cutii. Probabilitatea ca bila să aterizeze în oricare dintre cutiile date este $1/b$. Astfel, procesul de aruncare a bilelor este o secvenţă de probe bernoulliene cu o probabilitate de succes de $1/b$, unde succes înseamnă căderea bilei în cutia dată. Se poate formula o varietate de întrebări interesante referitoare la procesul de aruncare a bilelor.

Câte bile cad într-o cutie dată? Numărul de bile care cad într-o cutie dată urmează distribuţia binomială $b(k; n, 1/b)$. Dacă se aruncă n bile, numărul mediu de bile care cad în cutia dată este n/b.

Cât de multe bile trebuie aruncate, în medie, până când o cutie dată conţine o bilă? Numărul de aruncări, până când cutia dată conţine o bilă, urmează distribuţia geometrică cu probabilitatea $1/b$ şi astfel numărul mediu de aruncări, până se obţine un succes, este $1/(1/b) = b$.

Câte bile trebuie aruncate până când fiecare cutie conţine cel puţin o bilă? Să numim "lovitură" o aruncare în care o bilă cade într-o cutie goală. Dorim să ştim numărul mediu n de aruncări necesare pentru a obţine b lovituri.

Loviturile se pot utiliza pentru a partiţiona cele n aruncări în stadii. Al i-lea stadiu constă din aruncările de după a $(i-1)$-a lovitură până la a i-a lovitură. Primul stadiu constă din prima aruncare, deoarece ni se garantează că avem o lovitură atunci când toate cutiile sunt goale. Pentru fiecare aruncare din timpul celui de-al i-lea stadiu, există $i-1$ cutii ce conţin bile şi $b-i+1$ cutii goale. Astfel, pentru toate aruncările din stadiul i, probabilitatea de a obţine o lovitură este $(b-i+1)/b$.

Fie n_i numărul de aruncări la al i-lea stadiu. Astfel, numărul de aruncări necesare pentru a obţine b lovituri este $n = \sum_{i=1}^{b} n_i$. Fiecare variabilă aleatoare n_i are o distribuţie geometrică cu probabilitatea de succes $(b-i+1)/b$ şi de aceea

$$E[n_i] = \frac{b}{b-i+1}.$$

Din liniaritatea mediei, obţinem

$$E[n] = E\left[\sum_{i=1}^{b} n_i\right] = \sum_{i=1}^{b} E[n_i] = \sum_{i=1}^{b} \frac{b}{b-i+1} = b\sum_{i=1}^{b} \frac{1}{i} = b(\ln b + O(1)).$$

Ultima inegalitate rezultă din marginea (3.5) referitoare la seria armonică. Deci sunt necesare aproximativ $b \ln b$ aruncări înainte de a ne putea aştepta ca fiecare cutie să conţină o bilă.

6.6.3. Linii (secvenţe)

Să presupunem că aruncăm o monedă perfectă de n ori. Care este cea mai lungă linie (secvenţă) de steme consecutive pe care ne aşteptăm s-o obţinem? Răspunsul este $\Theta(\lg n)$, aşa cum ne arată analiza următoare.

Vom demonstra, întâi, că lungimea celei mai lungi secvenţe de steme este $O(\lg n)$. Fie A_{ik} evenimentul ca o secvenţă de steme de lungime cel puţin k să înceapă de la a i-a aruncare sau, mai precis, evenimentul ca cele k aruncări consecutive, $i, i+1, \ldots, i+k-1$ să ne dea numai steme, unde $1 \le k \le n$ şi $1 \le i \le n-k+1$. Pentru un eveniment dat A_{ik}, probabilitatea ca toate cele k aruncări să aibă ca rezultat stema, are o distribuţie geometrică cu $p = q = 1/2$:

$$\Pr\{A_{ik}\} = 1/2^k. \tag{6.47}$$

Pentru $k = 2\lceil \lg n \rceil$,

$$\Pr\left\{A_{i,2\lceil \lg n \rceil}\right\} = 1/2^{2\lceil \lg n \rceil} \le 1/2^{2 \lg n} = 1/n^2,$$

şi, astfel, probabilitatea ca o secvenţă de steme de lungime cel puţin $\lceil 2 \lg n \rceil$ să înceapă din poziţia i este chiar mică, considerând că sunt cel mult n poziţii (de fapt $n - 2\lceil \lg n \rceil + 1$) în care secvenţa ar putea începe. Probabilitatea ca o secvenţă de steme de lungime cel puţin $\lceil 2 \lg n \rceil$ să înceapă oriunde este deci

$$\Pr\left\{\bigcup_{i=1}^{n-2\lceil \lg n \rceil+1} A_{i,2\lceil \lg n \rceil}\right\} \le \sum_{i=1}^{n} 1/n^2 = \frac{1}{n},$$

deoarece, conform inegalităţii lui Boole (6.22), probabilitatea unei reuniuni de evenimente este cel mult suma probabilităţilor evenimentelor individuale. (De notat că inegalitatea lui Boole are loc şi pentru evenimente care nu sunt independente.)

Probabilitatea ca orice secvenţă să aibă lungimea cel puţin $2\lceil \lg n \rceil$ este cel mult $1/n$; deci probabilitatea ca cea mai lungă secvenţă să aibă lungimea mai mică decât $2\lceil \lg n \rceil$ este cel puţin $1 - 1/n$. Deoarece fiecare secvenţă are lungimea cel mult n, lungimea medie a celei mai lungi secvenţe este mărginită superior de

$$(\lceil 2 \lg n \rceil)(1 - 1/n) + n(1/n) = O(\lg n).$$

Şansa ca o secvenţă de steme să depăşească $r\lceil \lg n \rceil$ se diminuează rapid o dată cu r. Pentru $r \ge 1$, probabilitatea ca o secvenţă de $r\lceil \lg n \rceil$ să înceapă în poziţia i este

$$\Pr\{A_{i,r\lceil \lg n \rceil}\} = 1/2^{r\lceil \lg n \rceil} \le 1/n^r.$$

Astfel, probabilitatea ca cea mai lungă secvenţă să aibă lungimea cel puţin $r\lceil \lg n \rceil$ este cel mult $n/n^r = 1/n^{r-1}$, sau echivalent, probabilitatea ca cea mai lungă secvenţă să aibă lungimea mai mică decât $r\lceil \lg n \rceil$ este cel puţin $1 - 1/n^{r-1}$.

De exemplu, pentru $n = 1000$ de aruncări de monedă, probabilitatea de a avea o secvenţă de cel puţin $2\lceil \lg n \rceil = 20$ de steme este de cel mult $1/n = 1/1000$. Şansa de a avea o secvenţă mai lungă de $3\lceil \lg n \rceil = 30$ de steme este de cel mult $1/n^2 = 1/1000000$.

Vom demonstra acum o margine inferioară complementară: lungimea celei mai lungi secvenţe de steme în n aruncări este $\Omega(\lg n)$. Pentru a demonstra această margine vom căuta o secvenţă de lungime $\lfloor \lg n \rfloor / 2$. Din ecuaţia (6.47) avem

$$\Pr\{A_{i,\lfloor \lg n \rfloor/2}\} = 1/2^{\lfloor \lg n \rfloor/2} \ge 1/\sqrt{n}.$$

Probabilitatea ca o secvenţă de lungime cel puţin $\lfloor \lg n \rfloor / 2$ să nu înceapă în poziţia i este deci cel mult $1 - 1/\sqrt{n}$. Putem partiţiona cele n aruncări ale monedei în cel puţin $\lfloor 2n/\lfloor \lg n \rfloor \rfloor$ grupuri

de $\lfloor \lg n \rfloor / 2$ aruncări consecutive de monede. Deoarece aceste grupuri sunt formate din aruncări mutual exclusive şi independente, probabilitatea ca nici unul dintre aceste grupuri să nu fie o secvenţă de lungime $\lfloor \lg n \rfloor / 2$ este

$$(1 - 1/\sqrt{n})^{\lfloor 2n/\lfloor \lg n \rfloor \rfloor} \leq (1 - 1/\sqrt{n})^{2n/\lg n - 1} \leq e^{-(2n/\lg n - 1)/\sqrt{n}} = O(e^{-\lg n}) = O(1/n).$$

Pentru a justifica aceasta, am utilizat inegalitatea (2.7), $1 + x \leq e^x$. Astfel, probabilitatea ca cea mai lungă linie să depăşească $\lfloor \lg n \rfloor / 2$ este cel puţin $1 - O(1/n)$. Deoarece cea mai lungă secvenţă are lungimea cel puţin 0, lungimea medie a celei mai lungi linii (secvenţe) este de cel puţin

$$(\lfloor \lg n \rfloor / 2)(1 - O(1/n)) + 0 \cdot (1/n) = \Omega(\lg n).$$

Exerciţii

6.6-1 Presupunem că mai multe bile sunt aruncate în b cutii. Fiecare aruncare este independentă şi fiecare bilă are aceeaşi probabilitate de a ajunge în oricare dintre cutii. Care este numărul mediu de bile aruncate, înainte ca cel puţin una din cutii să conţină două bile?

6.6-2 ⋆ Pentru analiza paradoxului zilei de naştere, este important ca zilele de naştere să fie independente în totalitate (mutual independente) sau este suficient să fie independente două câte două? Justificaţi răspunsul.

6.6-3 ⋆ Câţi oameni trebuie invitaţi la o petrecere pentru a face verosimilă existenţa a *trei* dintre ei cu aceeaşi zi de naştere?

6.6-4 ⋆ Care este probabilitatea ca un k-şir peste o mulţime de lungime n să fie o k-permutare? Ce legătură este între această chestiune şi paradoxul zilei de naştere?

6.6-5 ⋆ Presupunem că n bile sunt aruncate în n cutii, fiecare aruncare este independentă şi fiecare bilă are aceeaşi probabilitate de a nimeri în oricare dintre cutii. Care este numărul mediu de cutii goale? Care este numărul mediu de cutii cu exact o bilă?

6.6-6 ⋆ Îmbunătăţiţi marginea inferioară a lungimii unei secvenţe arătând că, pentru n aruncări ale unei monede perfecte, probabilitatea ca să nu apară nici o linie de mai mult de $\lg n - 2 \lg \lg n$ steme consecutive este mai mică decât $1/n$.

Probleme

6-1 Bile şi cutii

În această problemă investigăm efectul diverselor ipoteze asupra numărului de moduri de a plasa n bile în b cutii distincte.

a. Să presupunem că cele n bile sunt distincte şi ordinea lor într-o cutie nu contează. Argumentaţi că numărul de moduri în care se pot aşeza bilele în cutii este b^n.

b. Să presupunem că bilele sunt distincte şi că bilele din fiecare cutie sunt ordonate. Demonstraţi că numărul de moduri în care se pot aşeza bilele în cutii este $(b+n-1)!/(b-1)!$. (*Indicaţie:* Consideraţi numărul de moduri de aranjare a n bile distincte şi $b-1$ beţişoare nedistinctibile într-un rând.)

c. Presupunem că bilele sunt identice şi că ordinea lor în cutie nu contează. Arătaţi că numărul de moduri în care se pot aşeza bilele în cutii este $\binom{b+n-1}{n}$. (*Indicaţie:* câte dintre aranjamentele din partea (b) se repetă dacă bilele devin identice?)

d. Presupunem că bilele sunt identice şi că nici o cutie nu conţine mai mult de o bilă. Arătaţi că numărul de moduri în care se pot aşeza bilele este $\binom{b}{n}$.

e. Presupunem că bilele sunt identice şi că nici o cutie nu poate rămâne goală. Arătaţi că numărul de moduri în care se pot aşeza bilele este $\binom{n-1}{b-1}$.

6-2 Analiza programului de determinare a maximului

Programul următor determină valoarea maximă dintr-un tablou neordonat $A[1..n]$.

```
1: max ← −∞
2: pentru i ← 1, n execută
3:    ▷ Compară A[i] cu max
4:    dacă A[i] > max atunci
5:       max ← A[i]
```

Dorim să determinăm numărul mediu de execuţii ale atribuirii din linia 5. Presupunem că numerele din A sunt o permutare aleatoare a n numere distincte.

a. Dacă un număr x este ales aleator dintr-o mulţime de i numere distincte, care este probabilitatea ca x să fie cel mai mare din mulţime?

b. Care este relaţia dintre $A[i]$ şi $A[j]$ când se execută linia 5 a programului, pentru $1 \leq j \leq i$?

c. Pentru orice i din domeniul $1 \leq i \leq n$, care este probabilitatea ca linia 5 să fie executată?

d. Fie $s_1, s_2, \ldots, s_n$ n variabile aleatoare, unde s_i reprezintă numărul de execuţii (0 sau 1) ale liniei 5 în timpul celei de-a i-a iteraţii a ciclului **pentru**. Ce este $\mathrm{E}[s_i]$?

e. Fie $s = s_1 + s_2 + \cdots + s_n$ numărul total de execuţii ale liniei 5 în timpul unei execuţii a programului. Arătaţi că $\mathrm{E}[s] = \Theta(\lg n)$.

6-3 Problema angajării

D-na profesoară Dixon vrea să angajeze un nou asistent cercetător. Ea a aranjat interviuri cu n solicitanţi şi vrea să-şi bazeze decizia doar pe calificarea lor. Din nefericire, regulile universităţii cer ca după fiecare interviu candidatul să fie respins sau să i se ofere postul solicitat. Profesoara Dixon a decis să adopte strategia de a selecta un întreg pozitiv $k < n$, să intervieveze şi să respingă primii k solicitanţi şi apoi să-l angajeze pe solicitantul care este mai pregătit decât toţi solicitanţii precedenţi. Dacă cel mai bun este între primii k intervievaţi, va fi angajat al n-lea solicitant. Arătaţi că d-na profesoară Dixon îşi maximizează şansa de a angaja cel mai bine pregătit solicitant, alegând k aproximativ egal cu n/e şi că şansa de a-l angaja pe cel mai bine pregătit este atunci aproximativ $1/e$.

6-4 Numărare probabilistică

Cu un contor de t biţi putem număra în mod obişnuit până la $2^t - 1$. Cu ***numărarea probabilistică*** a lui R. Morris putem număra până la valori mult mai mari, cu preţul unei pierderi a preciziei. Vom lua o valoare de contor i care va reprezenta numărul de n_i, pentru $i = 0, 1, \ldots, 2^t - 1$, unde n_i formează o secvenţă crescătoare de valori nenegative. Vom presupune că valoarea iniţială a contorului este 0, reprezentând numărul de valori $n_0 = 0$. Operaţia INCREMENTEAZĂ lucrează pe un contor ce conţine valoarea i în mod probabilist. Dacă $i = 2^t - 1$, se raportează o eroare de tip depăşire superioară. Altfel, contorul este incrementat cu 1 cu probabilitatea $1/(n_{i+1} - n_i)$ şi rămâne neschimbat cu probabilitatea $1 - 1/(n_{i+1} - n_i)$. Dacă selectăm $n_i = i$, pentru orice $i \geq 0$, atunci contorul este unul obişnuit. O situaţie mai interesantă apare dacă selectăm, de exemplu, $n_i = 2^{i-1}$, pentru $i > 0$, sau $n_i = F_i$ (al i-lea număr Fibonacci – vezi secţiunea 2.2). Pentru această problemă să presupunem că n_{2^t-1} este suficient de mare, astfel ca probabilitatea unei depăşiri superioare să fie neglijabilă.

a. Arătaţi că valoarea medie reprezentată de contor, după ce au fost efectuate n operaţii INCREMENTEAZĂ, este exact n.

b. Analiza dispersiei numărului reprezentat de contor depinde de secvenţa de numere n_i. Să considerăm un caz simplu: $n_i = 100i$, pentru orice $i \geq 0$. Estimaţi dispersia valorii reprezentate de registru după ce au fost efectuate n operaţii INCREMENTEAZĂ.

Note bibliografice

Primele metode generale de rezolvare a problemelor de probabilităţi au fost discutate în celebra corespondenţă dintre B. Pascal şi P. de Fermat, care a început în 1654 şi într-o carte a lui C. Huygens din 1657. Teoria riguroasă a probabilităţilor a început cu lucrarea lui J. Bernoulli din 1713 şi cu cea a lui A. de Moivre din 1730. Dezvoltări ulterioare ale teoriei au fost date de P. S. de Laplace, S.-D. Poisson şi C. F. Gauss.

Sumele de variabile aleatoare au fost studiate pentru prima dată de P. L. Chebyshev (Cebîşev) şi de A. A. Markov. Teoria probabilităţilor a fost axiomatizată de A. N. Kolmogorov în 1933. Marginile pentru cozile distribuţiilor au fost date de Chernoff [40] şi Hoeffding [99]. Lucrarea de bază referitoare la structurile aleatoare combinatorice a fost dată de P. Erdös.

Knuth [121] şi Liu [140] sunt referinţe bune pentru combinatorică elementară şi numărare. Cărţi standard ca Billingsley [28], Chung [41], Drake [57], Feller [66] şi Rozanov [171] dau o introducere comprehensivă în teoria probabilităţilor. Bollobás [30], Hofri [100] şi Spencer [179] conţin o mare varietate de tehnici probabilistice avansate.

II Ordonare şi statistici de ordine

Introducere

Această parte prezintă mai mulţi algoritmi care rezolvă urmatoarea ***problemă de ordonare***:

Intrare: Un şir de n numere $\langle a_1, a_2, ..., a_n \rangle$.

Ieşire: O permutare (reordonare) $\langle a'_1, a'_2, ..., a'_n \rangle$ a şirului dat astfel încât $a'_1 \leq \cdots \leq a'_n$.

Şirul de intrare este, de obicei, un tablou cu n elemente, deşi el poate fi reprezentat şi în alt mod, de exemplu sub forma unei liste înlănţuite.

Structura datelor

În practică, numerele care trebuie ordonate sunt rareori valori izolate. De obicei, fiecare număr face parte dintr-o colecţie de date numită ***articol***. Fiecare articol conţine o ***cheie***, care este valoarea ce trebuie ordonată, iar restul articolului conţine ***date adiţionale***, care sunt, de obicei, mutate împreună cu cheia. În practică, atunci când un algoritm de ordonare interschimbă cheile, el trebuie să interschimbe şi datele adiţionale. Dacă fiecare articol include o cantitate mare de date adiţionale, de multe ori interschimbăm un tablou de pointeri la articole, în loc să interschimbăm articolele înseşi, cu scopul de a minimiza cantitatea de date care este manevrată.

Dintr-un anumit punct de vedere, tocmai aceste detalii de implementare disting un algoritm de programul propriu-zis. Faptul că sortăm numere individuale sau articole mari, ce conţin numere, este irelevant pentru *metoda* prin care o procedură de ordonare determină ordinea elementelor. Astfel, atunci când ne concentrăm asupra problemei sortării, presupunem, de obicei, că intrarea constă numai din numere. Implementarea unui algoritm care sortează numere este directă din punct de vedere conceptual, deşi, într-o situaţie practică dată, pot exista şi alte subtilităţi care fac ca implementarea propriu-zisă a algoritmului să fie o sarcină dificilă.

Algoritmi de sortare

În capitolul 1 am prezentat doi algoritmi care ordonează n numere naturale. Sortarea prin inserţie necesită, în cazul cel mai defavorabil, un timp de ordinul $\Theta(n^2)$. Totuşi, datorită faptului că buclele sale interioare sunt mici, el este un algoritm rapid de sortare pe loc pentru dimensiuni mici ale datelor de intrare. (Reamintiţi-vă faptul că un algoritm de sortare sortează ***pe loc*** dacă doar un număr constant de elemente ale tabloului de intrare sunt stocate în afara tabloului la un moment dat.) Sortarea prin interclasare are un timp de execuţie asimptotic mai bun, $\Theta(n \lg n)$, dar procedura MERGE nu sortează în loc.

În această parte, vom introduce doi algoritmi noi de sortare pentru numere reale arbitrare. Heapsort, prezentat în capitolul 7, sortează pe loc n numere arbitrare într-un timp $O(n \lg n)$. Acest algoritm foloseşte o structură de date importantă, numită heap, pentru a implementa o coadă de prioritate.

Sortarea rapidă, prezentată în capitolul 8, sortează şi ea pe loc n numere, dar timpul său de execuţie, pentru cazul cel mai defavorabil, este $\Theta(n^2)$. Timpul mediu de execuţie este $\Theta(n \lg n)$ şi, de obicei, mai bun decât este, în practică, Heapsort. La fel ca sortarea prin inserţie, quicksort are un cod foarte compact, şi, deci, factorul constantei ascunse din timpul său de execuţie este mic. Sortarea rapidă este un algoritm pentru sortarea tablourilor de dimensiuni foarte mari.

Sortarea prin inserţie, sortarea prin interclasare, heapsort şi sortarea rapidă sunt toţi algoritmi de sortare prin comparare: ei determină ordinea elementelor unui tablou de intrare comparându-i elementele. Capitolul 9 începe prin introducerea modelului arborelui de decizie pentru studiul limitărilor de performanţă ale sortărilor prin comparare. Folosind acest model, vom demonstra o limită inferioară de $\Omega(n \lg n)$ pentru timpul de execuţie al celui mai defavorabil caz al oricărei sortări prin comparare a n elemente, demonstrând astfel că heapsort şi sortarea prin interclasare sunt sortări prin comparare, optime din punct de vedere asimptotic.

În capitolul 9, se arată, apoi, că putem învinge această limită inferioară de $\Omega(n \lg n)$ dacă putem aduna informaţii despre ordinea elementelor prin alte mijloace decât compararea elementelor. De exemplu, algoritmul de sortare prin numărare presupune că numerele de intrare fac parte din mulţimea $\{1, 2, ..., k\}$. Folosind indexarea tablourilor ca instrument pentru determinarea ordinii relative, sortarea prin numărare poate sorta n numere într-un timp de ordinul $O(k + n)$. Astfel, atunci când $k = O(n)$, sortarea prin numărare se execută într-un timp ce depinde, liniar, de numărul elementelor din tabloul de intrare. Un algoritm înrudit, radix sort, poate fi folosit pentru a extinde domeniul ordonării prin numărare. Dacă avem de ordonat n numere întregi, fiecare număr având d cifre, cu fiecare cifră aparţinând mulţimii $\{1, 2, ..., k\}$, sortarea pe baza cifrelor poate ordona numerele într-un timp de ordinul $O(d(n + k))$. Dacă d este constant şi k este $O(n)$, sortarea pe baza cifrelor se execută într-un timp liniar. Un al treilea algoritm, sortarea pe grupe, necesită informaţii despre distribuţia probabilistică a numerelor din tabloul de intrare. Algoritmul poate sorta n numere reale, distribuite uniform în intervalul semideschis $[0, 1)$, într-un timp mediu de ordinul $O(n)$.

Statistici de ordine

Statistica de ordinul i a unei mulţimi de n numere este al i-lea cel mai mic număr din mulţime. Desigur, o persoană poate determina statistica de ordinul i, sortând intrarea şi selectând elementul cu indicele i al ieşirii. Fără nici o presupunere despre distribuţia elementelor, această metodă rulează într-un timp de ordinul $\Omega(n \lg n)$, după cum arată şi limita inferioară determinată în capitolul 9.

În capitolul 10, vom arăta că putem determina al i-lea cel mai mic element în timpul $O(n)$, chiar şi atunci când elementele sunt numere reale arbitrare. Vom prezenta un algoritm având un pseudocod compact care se execută în timpul $O(n^2)$ în cazul cel mai defavorabil, dar într-un timp liniar în cazul general. Vom da, de asemenea, şi un algoritm mai complicat, care se execută într-un timp $O(n)$ în cazul cel mai defavorabil.

Fundamente

Deşi cele mai multe lucruri prezentate în această parte nu se bazează pe elemente matematice dificile, anumite secţiuni cer o abordare sofisticată din punct de vedere matematic. În special, analizele timpilor medii de execuţie ai algoritmilor de sortare rapida, sortare pe grupe, precum şi ai algoritmului bazat pe statistici de ordine folosesc elemente de probabilistică, elemente ce sunt prezentate în capitolul 6. Analiza timpului de execuţie al statisticii de ordine pentru cazul cel mai defavorabil necesită elemente matematice puţin mai sofisticate decât celelalte analize pentru timpii de execuţie în cazurile cele mai defavorabile din această parte.

7 Heapsort

În acest capitol vom prezenta un nou algoritm de sortare. Asemănător sortării prin interclasare, dar diferit de sortarea prin inserare, timpul de execuţie al agoritmului heapsort este $O(n \lg n)$. Asemănător sortării prin inserare şi diferit de sortarea prin interclasare, prin heapsort se ordonează elementele în spaţiul alocat vectorului: la un moment dat doar un număr constant de elemente ale vectorului sunt păstrate în afara spaţiului alocat vectorului de intrare. Astfel, algoritmul heapsort combină calităţile celor doi algoritmi de sortare prezentaţi deja.

Heapsort introduce o tehnică nouă de proiectare a algoritmilor bazată pe utilizarea unei structuri de date, numită de regulă ***heap***[1]. Structura de date heap este utilă nu doar pentru algoritmul heapsort, ea poate fi la fel de utilă şi în tratarea eficientă a unei cozi de prioritate. Cu structura de date heap ne vom mai întâlni în algoritmii din capitolele următoare.

Amintim că termenul heap a fost introdus şi utilizat iniţial în contextul algoritmului heapsort, dar acesta se foloseşte şi în legătură cu alocarea dinamică, respectiv în tratarea memoriei bazate pe "colectarea reziduurilor" (*garbage collected storage*), de exemplu în limbajele de tip Lisp. Structura de date heap ***nu*** se referă la heap-ul menţionat în alocarea dinamică, şi ori de câte ori, în această carte vom vorbi despre heap, vom înţelege structura definită în acest capitol.

7.1. Heap-uri

Structura de date ***heap (binar)*** este un vector care poate fi vizualizat sub forma unui arbore binar aproape complet (vezi secţiunea 5.5.3), conform figurii 7.1. Fiecare nod al arborelui corespunde unui element al vectorului care conţine valorile ataşate nodurilor. Arborele este plin, exceptând eventual nivelul inferior, care este plin de la stânga la dreapta doar până la un anumit loc. Un vector A care reprezintă un heap are două atribute: $lungime[A]$, reprezintă numărul elementelor din vector şi $dimensiune\text{-}heap[A]$ reprezintă numărul elementelor heap-ului memorat în vectorul A. Astfel, chiar dacă $A[1..lungime[A]]$ conţine în fiecare element al său date valide, este posibil ca elementele următoare elementului $A[dimensiune\text{-}heap[A]]$, unde $dimensiune\text{-}heap[A] \leq lungime[A]$, să nu aparţină heap-ului. Rădăcina arborelui este $A[1]$. Dat fiind un indice i, corespunzător unui nod, se pot determina uşor indicii părintelui acestuia PĂRINTE(i), al fiului STÂNGA(i) şi al fiului DREAPTA(i).

PĂRINTE(i)
  returnează $\lfloor i/2 \rfloor$

STÂNGA(i)
  returnează $2i$

DREAPTA(i)
  returnează $2i + 1$

[1] unii autori români, folosesc termenul "ansamblu".

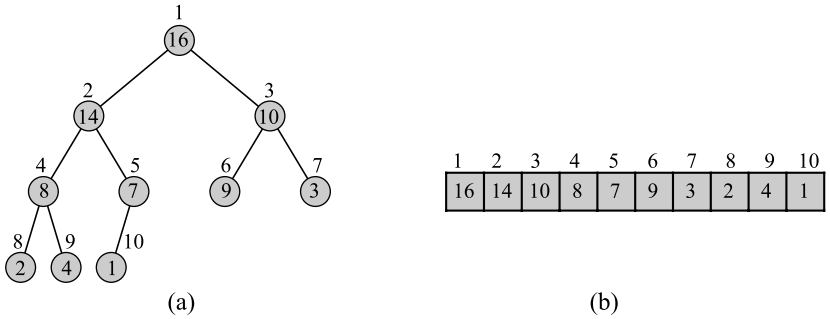


Figura 7.1 Un heap reprezentat sub forma unui arbore binar **(a)** şi sub forma unui vector **(b)**. Numerele înscrise în cercurile reprezentând nodurile arborelui sunt valorile ataşate nodurilor, iar cele scrise lângă cercuri sunt indicii elementelor corespunzătoare din vector.

În cele mai multe cazuri, procedura STÂNGA poate calcula valoarea $2i$ cu o singură instrucţiune, translatând reprezentarea binară a lui i la stânga cu o poziţie binară. Similar, procedura DREAPTA poate determina rapid valoarea $2i + 1$, translatând reprezentarea binară a lui i la stânga cu o poziţie binară, iar bitul nou intrat pe poziţia binară cea mai nesemnificativă va fi 1. În procedura PĂRINTE valoarea $\lfloor i/2 \rfloor$ se va calcula prin translatarea cu o poziţie binară la dreapta a reprezentării binare a lui i. Într-o implementare eficientă a algoritmului heapsort, aceste proceduri sunt adeseori codificate sub forma unor "macro-uri" sau a unor proceduri "in-line".

Pentru orice nod i, diferit de rădăcină, este adevărată următoarea ***proprietate de heap***:

$$A[\text{PĂRINTE}(i)] \geq A[i], \tag{7.1}$$

adică valoarea ataşată nodului este mai mică sau egală cu valoarea asociată părintelui său. Astfel cel mai mare element din heap este păstrat în rădăcină, iar valorile nodurilor oricărui subarbore al unui nod sunt mai mici sau egale cu valoarea nodului respectiv.

Definim ***înălţimea*** unui nod al arborelui ca fiind numărul muchiilor aparţinând celui mai lung drum care leagă nodul respectiv cu o frunză, iar înălţimea arborelui ca fiind înălţimea rădăcinii. Deoarece un heap având n elemente corespunde unui arbore binar complet, înălţimea acestuia este $\Theta(\lg n)$ (vezi exerciţiul 7.1-2). Vom vedea că timpul de execuţie al operaţiilor de bază, care se efectuează pe un heap, este proporţional cu înălţimea arborelui şi este $O(\lg n)$. În cele ce urmează, vom prezenta cinci proceduri şi modul lor de utilizare în algoritmul de sortare, respectiv într-o structură de tip coadă de prioritate.

- Procedura RECONSTITUIE-HEAP are timpul de execuţie $O(\lg n)$ şi este de primă importanţă în întreţinerea proprietăţii de heap (7.1).
- Procedura CONSTRUIEŞTE-HEAP are un timp de execuţie liniar şi generează un heap dintr-un vector neordonat, furnizat la intrare.
- Procedura HEAPSORT se execută în timpul $O(n \lg n)$ şi ordonează un vector în spaţiul alocat acestuia.
- Procedurile EXTRAGE-MAX şi INSEREAZĂ se execută în timpul $O(\lg n)$, iar cu ajutorul lor se poate utiliza un heap în realizarea unei cozi de prioritate.

Exerciţii

7.1-1 Care este cel mai mic, respectiv cel mai mare număr de elemente dintr-un heap având înălţimea h?

7.1-2 Arătaţi că un heap având n elemente are înălţimea $\lfloor \lg n \rfloor$.

7.1-3 Arătaţi că în orice subarbore al unui heap, rădăcina subarborelui conţine cea mai mare valoare care apare în acel subarbore.

7.1-4 Unde se poate afla cel mai mic element al unui heap, presupunând că toate elementele sunt distincte?

7.1-5 Este vectorul în care elementele se succed în ordine descrescătoare un heap?

7.1-6 Este secvenţa $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ un heap?

7.2. Reconstituirea proprietăţii de heap

Procedura RECONSTITUIE-HEAP este un subprogram important în prelucrarea heap-urilor. Datele de intrare ale acesteia sunt un vector A şi un indice i din vector. Atunci când se apelează RECONSTITUIE-HEAP, se presupune că subarborii, având ca rădăcini nodurile STÂNGA(i) respectiv DREAPTA(i), sunt heap-uri. Dar, cum elementul $A[i]$ poate fi mai mic decât descendenţii săi, acesta nu respectă proprietatea de heap (7.1). Sarcina procedurii RECONSTITUIE-HEAP este de a "scufunda" în heap valoarea $A[i]$, astfel încât subarborele care are în rădăcină valoarea elementului de indice i, să devină un heap.

RECONSTITUIE-HEAP(A, i)
1: $l \leftarrow$ STÂNGA(i)
2: $r \leftarrow$ DREAPTA(i)
3: **dacă** $l \leq$ *dimesiune-heap*$[A]$ şi $A[l] > A[i]$ **atunci**
4: $\quad maxim \leftarrow l$
5: **altfel**
6: $\quad maxim \leftarrow i$
7: **dacă** $r \leq$ *dimesiune-heap*$[A]$ şi $A[r] > A[maxim]$ **atunci**
8: $\quad maxim \leftarrow r$
9: **dacă** $maxim \neq i$ **atunci**
10: $\quad$ schimbă $A[i] \leftrightarrow A[maxim]$
11: $\quad$ RECONSTITUIE-HEAP($A, maxim$)

Figura 7.2 ilustrează efectul procedurii RECONSTITUIE-HEAP. La fiecare pas se determină cel mai mare element dintre $A[i]$, $A[$STÂNGA(i)$]$ şi $A[$DREAPTA(i)$]$, iar indicele său se păstrează în variabila *maxim*. Dacă $A[i]$ este cel mai mare, atunci subarborele având ca rădăcină nodul i este un heap şi procedura se termină. În caz contrar, cel mai mare element este unul dintre cei doi descendenţi şi $A[i]$ este interschimbat cu $A[maxim]$. Astfel, nodul i şi descendenţii săi satisfac proprietatea de heap. Nodul *maxim* are acum valoarea iniţială a lui $A[i]$, deci este posibil ca

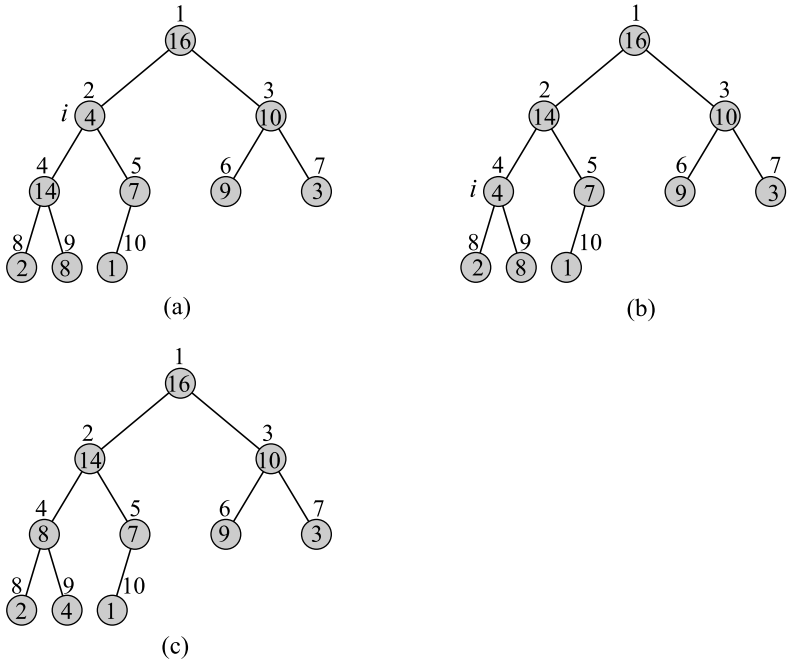


Figura 7.2 Efectul procedurii RECONSTITUIE-HEAP$(A, 2)$, unde *dimesiune-heap*$[A] = 10$. **(a)** Configuraţia iniţială a heap-ului, unde $A[2]$ (pentru nodul $i = 2$), nu respectă proprietatea de heap deoarece nu este mai mare decât descendenţii săi. Proprietatea de heap este restabilită pentru nodul 2 în **(b)** prin interschimbarea lui $A[2]$ cu $A[4]$, ceea ce anulează proprietatea de heap pentru nodul 4. Apelul recursiv al procedurii RECONSTITUIE-HEAP$(A, 4)$ poziţionează valoarea lui i pe 4. După interschimbarea lui $A[4]$ cu $A[9]$, aşa cum se vede în **(c)**, nodul 4 ajunge la locul său şi apelul recursiv RECONSTITUIE-HEAP$(A, 9)$ nu mai găseşte elemente care nu îndeplinesc proprietatea de heap.

subarborele de rădăcină *maxim* să nu îndeplinească proprietatea de heap. Rezultă că procedura RECONSTITUIE-HEAP trebuie apelată recursiv din nou pentru acest subarbore.

Timpul de execuţie al procedurii RECONSTITUIE-HEAP, corespunzător unui arbore de rădăcină i şi dimensiune n, este $\Theta(1)$, timp în care se pot analiza relaţiile dintre $A[i]$, $A[\text{STÂNGA}(i)]$ şi $A[\text{DREAPTA}(i)]$ la care trebuie adăugat timpul în care RECONSTITUIE-HEAP se execută pentru subarborele având ca rădăcină unul dintre descendenţii lui i. Dimensiunea acestor subarbori este de cel mult $2n/3$ – cazul cel mai defavorabil fiind acela în care nivelul inferior al arborelui este plin exact pe jumătate – astfel, timpul de execuţie al procedurii RECONSTITUIE-HEAP poate fi descris prin următoarea inegalitate recursivă:

$$T(n) \leq T(2n/3) + \Theta(1).$$

Soluţia acestei recurenţe se obţine pe baza celui de-al doilea caz al teoremei master (teorema 4.1): $T(n) = O(\lg n)$. Timpul de execuţie al procedurii RECONSTITUIE-HEAP pentru un nod de înălţime h poate fi exprimat alternativ ca fiind egal cu $O(h)$.

Exerciţii

7.2-1 Utilizând ca model figura 7.2, ilustraţi modul de funcţionare al procedurii RECONSTITUIE-HEAP$(A, 3)$ pentru vectorul $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$.

7.2-2 Care este efectul apelului procedurii RECONSTITUIE-HEAP(A, i), dacă elementul $A[i]$ este mai mare decât descendenţii săi?

7.2-3 Care este efectul procedurii RECONSTITUIE-HEAP(A, i) pentru $i >$ *dimesiune-heap*$[A]/2$?

7.2-4 Procedura RECONSTITUIE-HEAP poate fi implementată eficient din punct de vedere al timpului constant de execuţie, dar se poate întâmpla ca anumite compilatoare să genereze cod ineficient pentru apelul recursiv din linia a 11-a. Elaboraţi o variantă a procedurii RECONSTITUIE-HEAP, în care recursivitatea este înlocuită cu iterativitate.

7.2-5 Arătaţi că timpul de execuţie a procedurii RECONSTITUIE-HEAP, în cazul unui heap de dimensiune n, este, în cel mai defavorabil caz, $\Omega(\lg n)$. (*Indicaţie:* În cazul unui heap format din n elemente, determinaţi valorile ataşate nodurilor astfel încât procedura RECONSTITUIE-HEAP să fie apelată recursiv pentru toate nodurile aparţinând drumurilor care pornesc de la rădăcină la frunze.)

7.3. Construirea unui heap

Procedura RECONSTITUIE-HEAP poate fi utilizată "de jos în sus" pentru transformarea vectorului $A[1..n]$ în heap, unde $n = lungime[A]$. Deoarece toate elementele subşirului $A[(\lfloor n/2\rfloor + 1)..n]$ sunt frunze, acestea pot fi considerate ca fiind heap-uri formate din câte un element. Astfel, procedura CONSTRUIEŞTE-HEAP trebuie să traverseze doar restul elementelor şi să execute procedura RECONSTITUIE-HEAP pentru fiecare nod întâlnit. Ordinea de prelucrare a nodurilor asigură ca subarborii, având ca rădăcină descendenţi ai nodului i să formeze heap-uri înainte ca RECONSTITUIE-HEAP să fie executat pentru aceste noduri.

CONSTRUIEŞTE-HEAP(A)
1: $dimesiune\text{-}heap[A] \leftarrow lungime[A]$
2: **pentru** $i \leftarrow \lfloor lungime[A]/2\rfloor$,1 **execută**
3: RECONSTITUIE-HEAP(A, i)

Figura 7.3 ilustrează modul de funcţionare al procedurii CONSTRUIEŞTE-HEAP.

Timpul de execuţie al procedurii CONSTRUIEŞTE-HEAP poate fi calculat simplu, determinând limita superioară a acestuia: fiecare apel al procedurii RECONSTITUIE-HEAP necesită un timp $O(\lg n)$ şi, deoarece pot fi $O(n)$ asemenea apeluri, rezultă că timpul de execuţie poate fi cel mult $O(n \lg n)$. Această estimare este corectă, dar nu este suficient de tare asimptotic.

Vom obţine o limită mai tare observând că timpul de execuţie a procedurii RECONSTITUIE-HEAP depinde de înălţimea nodului în arbore, aceasta fiind mică pentru majoritatea nodurilor. Estimarea noastră mai riguroasă se bazează pe faptul că un heap având n elemente are înălţimea $\lg n$ (vezi exerciţiul 7.1-2) şi că pentru orice înălţime h, în heap există cel mult $\lfloor n/2^{h+1}\rfloor$ noduri de înălţime h (vezi exerciţiul 7.3-3).

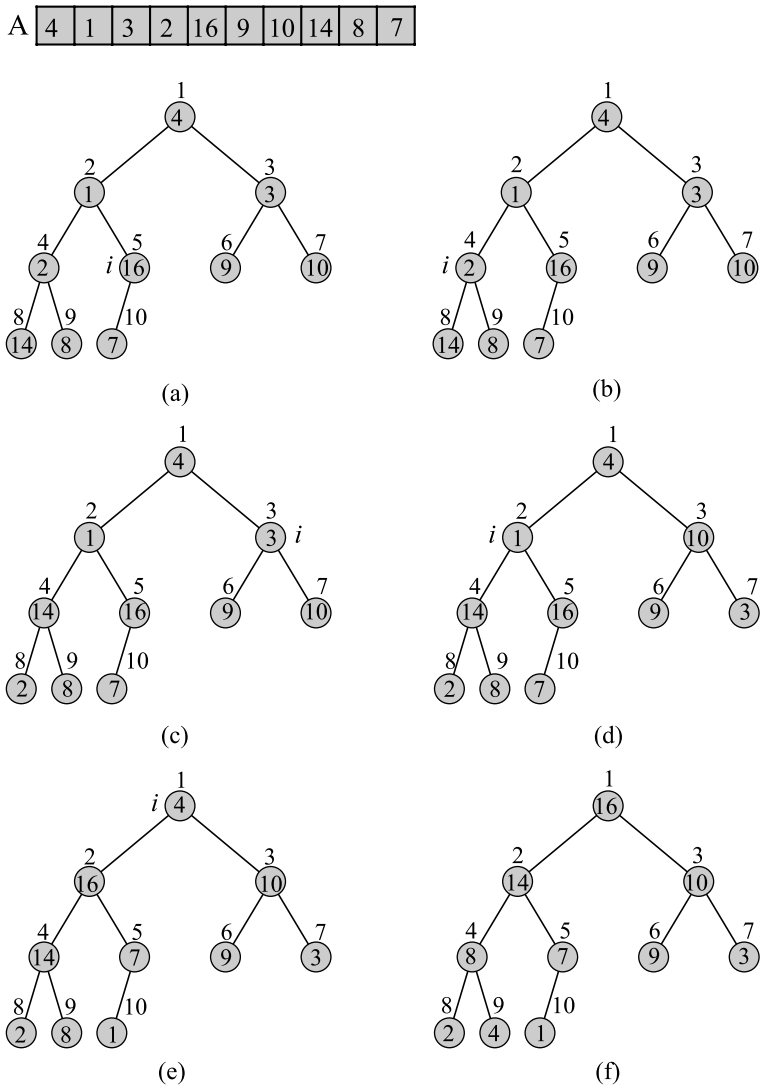


Figura 7.3 Modul de execuţie a procedurii CONSTRUIEŞTE-HEAP. În figură se vizualizează structurile de date în starea lor anterioară apelului procedurii RECONSTITUIE-HEAP (linia 3 din procedura CONSTRUIEŞTE-HEAP). **(a)** Se consideră un vector A având 10 elemente şi arborele binar corespunzător. După cum se vede în figură, variabila de control i a ciclului, în momentul apelului RECONSTITUIE-HEAP(A, i), indică nodul 5. **(b)** reprezintă rezultatul; variabila de control i a ciclului acum indică nodul 4. **(c) - (e)** vizualizează iteraţiile succesive ale ciclului **pentru** din CONSTRUIEŞTE-HEAP. Se observă că, atunci când se apelează procedura RECONSTITUIE-HEAP pentru un nod dat, subarborii acestui nod sunt deja heap-uri. **(f)** reprezintă heap-ul final al procedurii CONSTRUIEŞTE-HEAP.

Timpul de execuţie a procedurii RECONSTITUIE-HEAP pentru un nod de înălţime h fiind $O(h)$, obţinem pentru timpul de execuţie a procedurii CONSTRUIEŞTE-HEAP:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right). \tag{7.2}$$

Ultima însumare poate fi evaluată prin înlocuirea $x = 1/2$ în formula (3.6). Obţinem

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2. \tag{7.3}$$

Astfel, timpul de execuţie al procedurii CONSTRUIEŞTE-HEAP poate fi estimat ca fiind:

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n). \tag{7.4}$$

De aici rezultă că se poate construi un heap dintr-un vector într-un timp liniar.

Exerciţii

7.3-1 Utilizând ca model figura 7.3, ilustraţi modul de funcţionare al procedurii CONSTRUIEŞTE-HEAP pentru vectorul $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

7.3-2 De ce trebuie micşorată variabila de control i a ciclului în linia 2 a procedurii CONSTRUIEŞTE-HEAP de la $\lfloor lungime[A]/2 \rfloor$ la 1, în loc să fie mărită de la 1 la $\lfloor lungime[A]/2 \rfloor$?

7.3-3 $\star$ Arătaţi că, într-un heap de înălţime h având n elemente, numărul nodurilor este cel mult $\lceil n/2^{h+1} \rceil$.

7.4. Algoritmul heapsort

Algoritmul heapsort începe cu apelul procedurii CONSTRUIEŞTE-HEAP în scopul transformării vectorului de intrare $A[1..n]$ în heap, unde $n = lungime[A]$. Deoarece cel mai mare element al vectorului este ataşat nodului rădăcină $A[1]$, acesta va ocupa locul definitiv în vectorul ordonat prin interschimbarea sa cu $A[n]$. În continuare, "excluzând" din heap cel de-al n-lea element (şi micşorând cu 1 *dimesiune-heap*$[A]$), restul de $A[1..(n-1)]$ elemente se pot transforma uşor în heap, deoarece subarborii nodului rădăcină au proprietatea de heap (7.1), cu eventuala excepţie a elementului ajuns în nodul rădăcină.

HEAPSORT(A)
1: CONSTRUIEŞTE-HEAP(A)
2: **pentru** $i \leftarrow lungime[A]$, 2 **execută**
3: schimbă $A[1] \leftrightarrow A[i]$
4: $dimesiune\text{-}heap[A] \leftarrow dimesiune\text{-}heap[A] - 1$
5: RECONSTITUIE-HEAP$(A, 1)$

Apelând procedura RECONSTITUIE-HEAP$(A, 1)$ se restabileşte proprietatea de heap pentru vectorul $A[1..(n-1)]$. Acest procedeu se repetă micşorând dimensiunea heap-ului de la $n-1$ la 2.

Figura 7.4 ilustrează, pe un exemplu, modul de funcţionare a procedurii HEAPSORT, după ce în prealabil datele au fost transformate în heap. Fiecare heap reprezintă starea iniţială la începutul pasului iterativ (linia 2 din ciclul **pentru**).

Timpul de execuţie al procedurii HEAPSORT este $O(n \lg n)$, deoarece procedura CONSTRUIEŞTE-HEAP se execută într-un timp $O(n)$, iar procedura RECONSTITUIE-HEAP, apelată de $n-1$ ori, se execută în timpul $O(\lg n)$.

Exerciţii

7.4-1 Utilizând ca model figura 7.4, ilustraţi modul de funcţionare al procedurii HEAPSORT pentru vectorul $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

7.4-2 Care este timpul de execuţie al algoritmului heapsort în cazul unui vector A de dimensiune n, ordonat crescător? Dar în cazul unui vector ordonat descrescător?

7.4-3 Arătaţi că timpul de execuţie al algoritmului heapsort este $\Omega(n \lg n)$.

7.5. Cozi de priorităţi

Heapsort este un algoritm excelent, dar o implementare bună a algoritmului de sortare rapidă, algoritm prezentat în capitolul 8, se poate dovedi de multe ori mai eficientă. În schimb, structura de date heap este foarte utilă. În această secţiune vom prezenta una din cele mai frecvente aplicaţii ale unui heap: utilizarea lui sub forma unei cozi de priorităţi.

Coada de priorităţi este o structură de date care conţine o mulţime S de elemente, fiecare având asociată o valoare numită ***cheie***. Asupra unei cozi de priorităţi se pot efectua următoarele operaţii.

INSEREAZĂ(S, x) inserează elementul x în mulţimea S. Această operaţie poate fi scrisă în felul următor: $S \leftarrow S \cup \{x\}$.

MAXIM(S) returnează elementul din S având cheia cea mai mare.

EXTRAGE-MAX(S) elimină şi returnează elementul din S având cheia cea mai mare.

O aplicaţie a cozilor de priorităţi constă în planificarea lucrărilor pe calculatoare partajate. Sarcinile care trebuie efectuate şi priorităţile relative se memorează într-o coadă de prioritate. Când o lucrare este terminată sau întreruptă, procedura EXTRAGE-MAX va selecta lucrarea având prioritatea cea mai mare dintre lucrările în aşteptare. Cu ajutorul procedurii INSERARE, în coadă poate fi introdusă oricând o lucrare nouă.

O coadă de priorităţi poate fi utilizată şi în simularea unor evenimente controlate. Din coadă fac parte evenimentele de simulat, fiecăruia ataşându-i-se o cheie reprezentând momentul când va avea loc evenimentul. Evenimentele trebuie simulate în ordinea desfăşurării lor în timp, deoarece simularea unui eveniment poate determina necesitatea simulării altuia în viitor. În cazul acestei aplicaţii, este evident că ordinea evenimentelor în coada de priorităţi trebuie inversată, iar

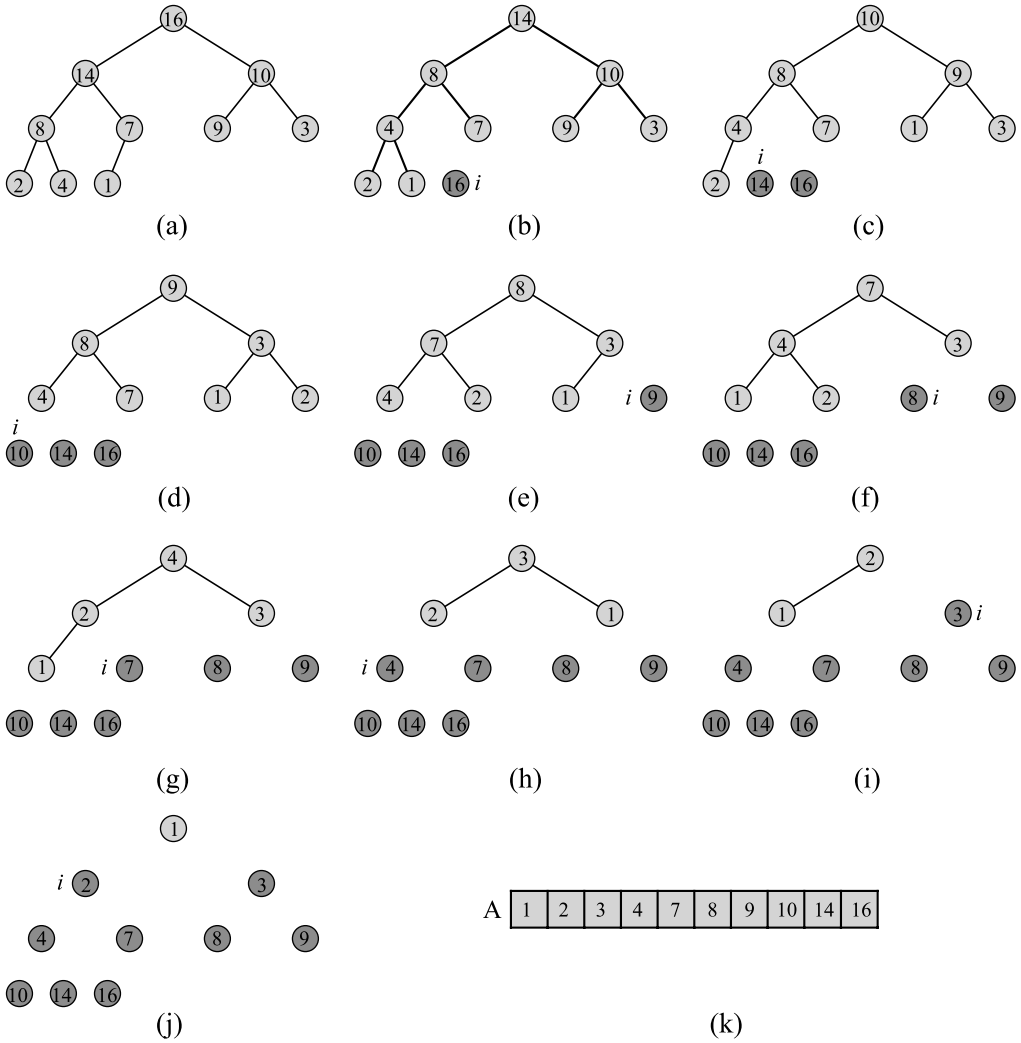


Figura 7.4 Modul de funcţionare a algoritmului HEAPSORT. **(a)** Structura de date heap, imediat după construirea sa de către procedura CONSTRUIEŞTE-HEAP. **(b)-(j)** Heap-ul, imediat după câte un apel al procedurii RECONSTITUIE-HEAP (linia 5 în algoritm). Figura reprezintă valoarea curentă a variabilei i. Din heap fac parte doar nodurile din cercurile nehaşurate. **(k)** Vectorul A ordonat, obţinut ca rezultat.

procedurile MAXIM şi EXTRAGE-MAX se vor înlocui cu MINIM şi EXTRAGE-MIN. Programul de simulare va determina următorul eveniment cu ajutorul procedurii EXTRAGE-MIN, iar dacă va trebui introdus un nou element în şir, se va apela procedura INSERARE.

Rezultă în mod firesc faptul că o coadă de priorităţi poate fi implementată utilizând un heap. Operaţia MAXIM-HEAP va determina în timpul $\Theta(1)$ cel mai mare element al heap-ului care, de fapt, este valoarea $A[1]$. Procedura EXTRAGE-MAX-DIN-HEAP este similară structurii repetitive **pentru** (liniile 3–5) din procedura HEAPSORT:

EXTRAGE-MAX-DIN-HEAP(A)
1: **dacă** *dimesiune-heap*$[A] < 1$ **atunci**
2: **eroare** "depăşire inferioară heap"
3: $max \leftarrow A[1]$
4: $A[1] \leftarrow A[$*dimesiune-heap*$[A]]$
5: *dimesiune-heap*$[A] \leftarrow$ *dimesiune-heap*$[A] - 1$
6: RECONSTITUIE-HEAP($A, 1$)
7: **returnează** *maxim*

Timpul de execuţie al procedurii EXTRAGE-MAX-DIN-HEAP este $O(\lg n)$, deoarece conţine doar câţiva paşi, care se execută în timp constant înainte să se execute procedura RECONSTITUIE-HEAP, care necesită un timp de $O(\lg n)$.

Procedura INSEREAZĂ-ÎN-HEAP inserează un nod în heap-ul A. La prima expandare a heap-ului se adaugă o frunză arborelui. Apoi, la fel ca în structura repetitivă de inserare (liniile 5–7) din SORTARE-PRIN-INSERARE (din secţiunea 1.1), se traversează un drum pornind de la această frunză către rădăcină, în scopul găsirii locului definitiv al noului element.

INSEREAZĂ-ÎN-HEAP(A, *cheie*)
1: *dimesiune-heap*$[A] \leftarrow$ *dimesiune-heap*$[A] + 1$
2: $i \leftarrow$ *dimesiune-heap*$[A]$
3: **cât timp** $i > 1$ şi $A[$PĂRINTE$(i)] <$ *cheie* **execută**
4: $A[i] \leftarrow A[$PĂRINTE$(i)]$
5: $i \leftarrow$ PĂRINTE(i)
6: $A[i] \leftarrow$ *cheie*

În figura 7.5 este ilustrat un exemplu al operaţiei INSEREAZĂ-ÎN-HEAP. Timpul de execuţie al procedurii INSEREAZĂ-ÎN-HEAP, pentru un heap având n elemente, este $O(\lg n)$, deoarece drumul parcurs de la noua frunză către rădăcină are lungimea $O(\lg n)$.

În concluzie, pe un heap se poate efectua orice operaţie specifică unei cozi de priorităţi, definită pe o mulţime având n elemente, într-un timp $O(\lg n)$.

Exerciţii

7.5-1 Utilizând ca model figura 7.5, ilustraţi operaţia INSEREAZĂ-ÎN-HEAP($A, 10$) pentru heap-ul $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

7.5-2 Dat fiind heap-ul $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$, ilustraţi operaţia EXTRAGE-MAX-DIN-HEAP.

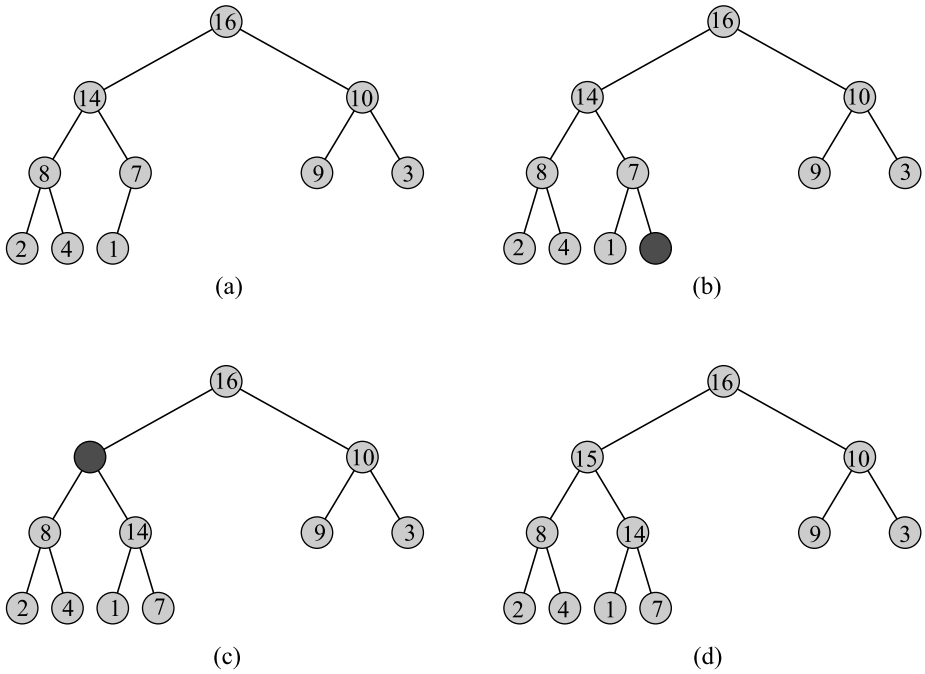


Figura 7.5 Operaţia INSEREAZĂ-ÎN-HEAP. **(a)** Heap-ul din figura 7.4(a) înainte de inserarea nodului având cheia 15. **(b)** Se adaugă arborelui o frunză nouă. **(c)** Se "scufundă" valorile de pe drumul dintre frunză şi rădăcină până la găsirea nodului corespunzător cheii 15. **(d)** Se inserează cheia 15.

7.5-3 Arătaţi cum se poate implementa o listă de tip FIFO cu ajutorul unei cozi de priorităţi. Arătaţi cum se poate implementa o stivă cu ajutorul unei cozi de priorităţi. (Listele FIFO şi stivele vor fi definite în secţiunea 11.1.)

7.5-4 Scrieţi o implementare a procedurii HEAP-CU-CHEI-CRESCĂTOARE(A, i, k) de timp $O(\lg n)$, care realizează atribuirea $A[i] \leftarrow \max(A[i], k)$, şi actualizează structura heap în mod corect.

7.5-5 Operaţia ŞTERGE-DIN-HEAP(A, i) şterge elementul ataşat nodului i din heap-ul A. Găsiţi o implementare pentru operaţia ŞTERGE-DIN-HEAP care se execută pentru un heap având n elemente într-un timp $O(\lg n)$.

7.5-6 Găsiţi un algoritm de timp $O(n \lg k)$ pentru a interclasa k liste ordonate, unde n este numărul total de elemente din listele de intrare. (*Indicaţie:* se utilizează un heap)

Probleme

7-1 Construirea unui heap prin inserare
Procedura CONSTRUIEŞTE-HEAP, din secţiunea 7.3, poate fi implementată folosind, în mod repetat, procedura INSERARE-ÎN-HEAP, în scopul inserării elementelor în heap. Fie următoarea implementare:

CONSTRUIEŞTE-HEAP$'(A)$
1: $dimesiune\text{-}heap[A] \leftarrow 1$
2: **pentru** $i \leftarrow 2$, $lungime[A]$ **execută**
3: INSEREAZĂ-ÎN-HEAP$(A, A[i])$

a. Pentru date de intrare identice, procedurile CONSTRUIEŞTE-HEAP şi CONSTRUIEŞTE-HEAP$'$ construiesc acelaşi heap? Demonstraţi sau găsiţi contraexemple.

b. Arătaţi că, în cazul cel mai defavorabil, procedura CONSTRUIEŞTE-HEAP$'$ cere un timp $\Theta(n \lg n)$ pentru a construi un heap având n elemente.

7-2 Analiza heap-urilor d-are
Un ***heap d-ar*** este asemănător unui heap binar, dar nodurile nu au doi ci d descendenţi.

a. Cum se poate reprezenta un heap d-ar sub forma unui vector?

b. Care este înălţimea (în funcţie de n şi d) a unui heap d-ar, având n elemente?

c. Descrieţi o implementare eficientă pentru operaţia EXTRAGE-MAX. Analizaţi timpul de execuţie în funcţie de n şi d.

d. Descrieţi o implementare eficientă pentru operaţia INSERARE. Analizaţi timpul de execuţie în funcţie de n şi d.

e. Descrieţi o implementare eficientă pentru operaţia HEAP-CU-CHEI-CRESCĂTOARE(A, i, k) care realizează $A[i] \leftarrow \max(A[i], k)$ şi actualizează structura heap în mod corect. Analizaţi timpul de execuţie în funcţie de d şi n.

Note bibliografice

Algoritmul heapsort a fost inventat de Williams [202], care a descris şi modul de implementare a unei cozi de priorităţi printr-un heap. Procedura CONSTRUIEŞTE-HEAP a fost propusă de Floyd [69].

8 Sortarea rapidă

Sortarea rapidă este un algoritm de sortare care, pentru un şir de n elemente, are un timp de execuţie $\Theta(n^2)$, în cazul cel mai defavorabil. În ciuda acestei comportări proaste, în cazul cel mai defavorabil, algoritmul de sortare rapidă este deseori cea mai bună soluţie practică, deoarece are o comportare medie remarcabilă: timpul său mediu de execuţie este $\Theta(n \lg n)$, şi constanta ascunsă în formula $\Theta(n \lg n)$ este destul de mică. Algoritmul are avantajul că sortează pe loc (în spaţiul alocat şirului de intrare) şi lucrează foarte bine chiar şi într-un mediu de memorie virtuală.

În secţiunea 8.1 sunt descrişi algoritmul şi un subalgoritm important, folosit de sortarea rapidă pentru partiţionare. Deoarece comportarea algoritmului de sortare rapidă este foarte complexă, vom începe studiul performanţei lui cu o discuţie intuitivă în secţiunea 8.2, şi lăsăm analiza precisă la sfârşitul capitolului. În secţiunea 8.3 sunt prezentate două variante ale algoritmului de sortare rapidă, care utilizează un generator de numere aleatoare. Aceşti algoritmi aleatori au multe proprietăţi interesante. Timpul lor mediu de execuţie este bun şi nu se cunosc date de intrare particulare pentru care să aibă comportarea cea mai proastă. O variantă aleatoare a algoritmului de sortare rapidă este studiată în secţiunea 8.4 şi se demonstrează că timpul lui de execuţie, în cazul cel mai defavorabil, este $O(n^2)$, iar timpul mediu $O(n \lg n)$.

8.1. Descrierea sortării rapide

Algoritmul de sortare rapidă, ca de altfel şi algoritmul de sortare prin interclasare, se bazează pe paradigma "divide şi stăpâneşte", introdusă în secţiunea 1.3.1. Iată un proces "divide şi stăpâneşte" în trei paşi, pentru un subşir $A[p..r]$.

Divide: Şirul $A[p..r]$ este împărţit (rearanjat) în două subşiruri nevide $A[p..q]$ şi $A[q+1..r]$, astfel încât fiecare element al subşirului $A[p..q]$ să fie mai mic sau egal cu orice element al subşirului $A[q+1..r]$. Indicele q este calculat de procedura de partiţionare.

Stăpâneşte: Cele două subşiruri $A[p..q]$ şi $A[q+1..r]$ sunt sortate prin apeluri recursive ale algoritmului de sortare rapidă.

Combină: Deoarece cele două subşiruri sunt sortate pe loc, nu este nevoie de nici o combinare, şirul $A[p..r]$ este ordonat.

Descrierea algoritmului este următoarea:

QUICKSORT(A, p, r)
1: **dacă** $p < r$ **atunci**
2: $q \leftarrow$ PARTIŢIE(A, p, r)
3: QUICKSORT(A, p, q)
4: QUICKSORT$(A, q+1, r)$

Pentru ordonarea întregului şir A, iniţial se apelează QUICKSORT(A, 1, *lungime*[A]).

Partiţionarea şirului

Cheia algoritmului este procedura PARTIŢIE, care rearanjează pe loc subşirul $A[p..r]$.

```
PARTIŢIE(A, p, r)
 1: x ← A[p]
 2: i ← p − 1
 3: j ← r + 1
 4: cât timp ADEVĂRAT execută
 5:    repetă
 6:       j ← j − 1
 7:    până când A[j] ≤ x
 8:    repetă
 9:       i ← i + 1
10:    până când A[i] ≥ x
11:    dacă i < j atunci
12:       interschimbă A[i] ↔ A[j]
13:    altfel
14:       returnează j
```

În figura 8.1 este ilustrat modul de funcţionare a procedurii PARTIŢIE. Întâi se selectează un element $x = A[p]$ din şirul $A[p..r]$, care va fi elementul "pivot", în jurul căruia se face partiţionarea şirului $A[p..r]$. Apoi, două subşiruri $A[p..i]$ şi $A[j..r]$ cresc la începutul şi respectiv, sfârşitul şirului $A[p..r]$, astfel încât fiecare element al şirului $A[p..i]$ să fie mai mic sau egal cu x, şi orice element al şirului $A[j..r]$, mai mare sau egal cu x. La început $i = p - 1$ şi $j = r + 1$, deci cele două subşiruri sunt vide.

În interiorul ciclului **cât timp**, în liniile 5–7, indicele j se decrementează, iar i se incrementează până când $A[i] \geq x \geq A[j]$. Presupunând că inegalităţile de mai sus sunt stricte, $A[i]$ este prea mare ca să aparţină primului subşir (cel de la început), iar $A[j]$ prea mic ca să aparţină celui de al doilea subşir (cel de la sfârşit). Astfel, interschimbând $A[i]$ cu $A[j]$ (linia 12), cele două părţi cresc. (Interschimbarea se poate face şi în cazul în care avem inegalităţi stricte.)

Ciclul **cât timp** se repetă până când inegalitatea $i \geq j$ devine adevărată. În acest moment, întregul şir $A[p..r]$ este partiţionat în două subşiruri $A[p..q]$ şi $A[q+1..r]$, astfel încât $p \leq q < r$ şi nici un element din $A[p..q]$ nu este mai mare decât orice element din $A[q+1..r]$. Procedura returnează valoarea $q = j$.

De fapt, procedura de partiţionare execută o operaţie simplă: pune elementele mai mici decât x în primul subşir, iar pe cele mai mari decât x în subşirul al doilea. Există câteva particularităţi care determină o comportare interesantă a procedurii PARTIŢIE. De exemplu, indicii i şi j nu depăşesc niciodată marginile vectorului $A[p..r]$, dar acest lucru nu se vede imediat din textul procedurii. Un alt exemplu: este important ca elementul $A[p]$ să fie utilizat drept element pivot x. În schimb, dacă se foloseşte $A[r]$ ca element pivot, şi, întâmplător, $A[r]$ este cel mai mare element al vectorului $A[p..r]$, atunci PARTIŢIE returnează procedurii QUICKSORT valoarea $q = r$, şi procedura intră într-un ciclu infinit. Problema 8-1 cere să se demonstreze corectitudinea procedurii PARTIŢIE.

Timpul de execuţie al procedurii PARTIŢIE, în cazul unui vector $A[p..r]$, este $\Theta(n)$, unde $n = r - p + 1$ (vezi exerciţiul 8.1-3.).

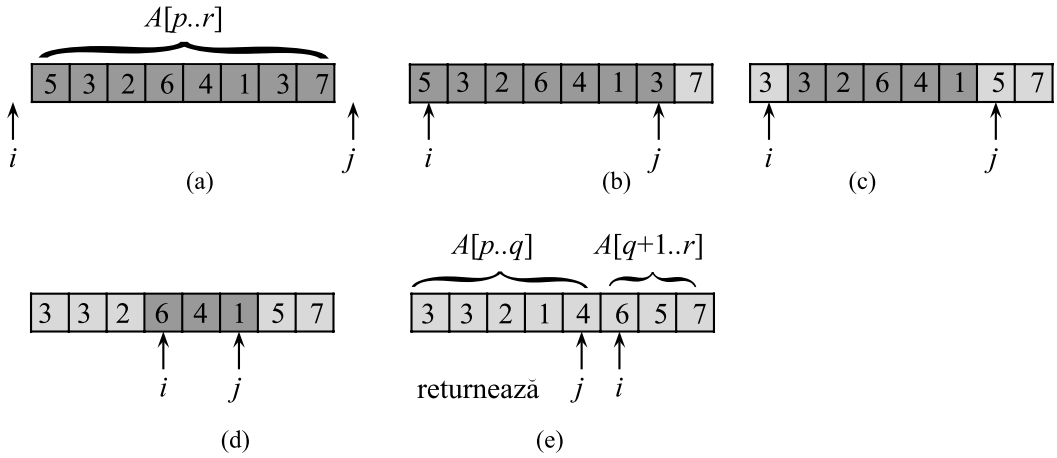


Figura 8.1 Operaţiile efectuate de procedura PARTIŢIE pe un exemplu. Elementele haşurate în gri deschis sunt deja plasate în poziţiile lor corecte, iar cele haşurate închis încă nu. (**a**) Şirul de intrare, cu valorile iniţiale ale variabilelor i şi j, care puncteză în afara şirului. Vom face partiţionarea în jurul elementului $x = A[p] = 5$. (**b**) Poziţiile lui i şi j în linia 11 a algoritmului, după prima parcurgere a ciclului **cât timp**. (**c**) Rezultatul schimbului de elemente descris în linia 12. (**d**) Valorile lui i şi j în linia 11 după a doua parcurgere a ciclului **cât timp**. (**e**) Valorile lui i şi j după a treia şi ultima iteraţie a ciclului **cât timp**. Procedura se termină deoarece $i \geq j$ şi valoarea returnată este $q = j$. Elementele şirului până la $A[j]$, inclusiv, sunt mai mici sau egale cu $x = 5$, iar cele de după $A[j]$, sunt toate mai mici sau egale cu $x = 5$.

Exerciţii

8.1-1 Folosind figura 8.1 drept model, să se ilustreze operaţiile procedurii PARTIŢIE în cazul vectorului $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

8.1-2 Ce valoare a lui q returnează procedura PARTIŢIE, dacă toate elementele vectorului $A[p..r]$ sunt egale.

8.1-3 Să se argumenteze, pe scurt, afirmaţia că timpul de execuţie al procedurii PARTIŢIE, pentru un vector de n elemente, este $\Theta(n)$.

8.1-4 Cum trebuie modificată procedura QUICKSORT pentru a ordona descrescător?

8.2. Performanţa algoritmului de sortare rapidă

Timpul de execuţie al algoritmului de sortare rapidă depinde de faptul că partiţionarea este echilibrată sau nu, iar acesta din urmă de elementele alese pentru partiţionare. Dacă partiţionarea este echilibrată, algoritmul asimptotic este la fel de rapid ca sortarea prin interclasare. În cazul în care partiţionarea nu este echilibrată, algoritmul se execută la fel de încet ca sortarea prin inserare. În această secţiune vom investiga, fără rigoare matematică, performanţa algoritmului de sortare rapidă în cazul partiţionării echilibrate.

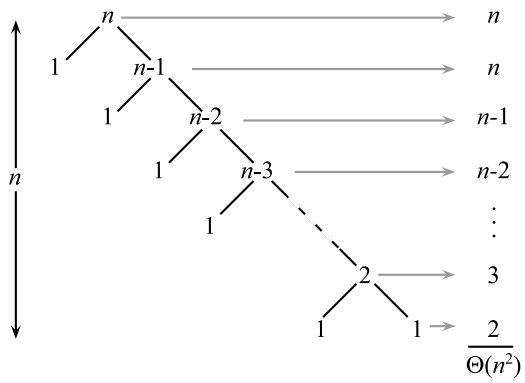


Figura 8.2 Arborele de recursivitate pentru QUICKSORT când procedura PARTIŢIE pune întotdeauna într-o parte a vectorului numai un singur element (cazul cel mai defavorabil). Timpul de execuţie în acest caz este $\Theta(n^2)$.

Partiţionarea în cazul cel mai defavorabil

Comportarea cea mai defavorabilă a algoritmului de sortare rapidă apare în situaţia în care procedura de partiţionare produce un vector de $n-1$ elemente şi unul de 1 element. (Demonstraţia se face în secţiunea 8.4.1) Să presupunem că această partiţionare dezechilibrată apare la fiecare pas al algoritmului. Deoarece timpul de partiţionare este de $\Theta(n)$, şi $T(1) = \Theta(1)$, formula recursivă pentru timpul de execuţie a algoritmului de sortare rapidă este:

$$T(n) = T(n-1) + \Theta(n).$$

Pentru evaluarea formulei de mai sus, observăm că $T(1) = \Theta(1)$, apoi iterăm formula:

$$T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2).$$

Ultima egalitate se obţine din observaţia că $\sum_{k=1}^{n} k$ este o progresie aritmetică (3.2). În figura 8.2 este ilustrat arborele de recursivitate pentru acest cel mai defavorabil caz al algoritmului de sortare rapidă. (Vezi secţiunea 4.2 pentru alte detalii privind arborii recursivi.)

Dacă partiţionarea este total dezechilibrată la fiecare pas recursiv al algoritmului, atunci timpul de execuţie este $\Theta(n^2)$. Deci timpul de execuţie, în cazul cel mai defavorabil, nu este mai bun decât al algoritmului de sortare prin inserare. Mai mult, timpul de execuţie este $\Theta(n^2)$ chiar şi în cazul în care vectorul de intrare este ordonat – caz în care algoritmul de sortare prin inserare are timpul de execuţie $O(n)$.

Partiţionarea în cazul cel mai favorabil

Dacă algoritmul de partiţionare produce doi vectori de $n/2$ elemente, algoritmul de sortare rapidă lucrează mult mai repede. Formula de recurenţă în acest caz este:

$$T(n) = 2T(n/2) + \Theta(n)$$

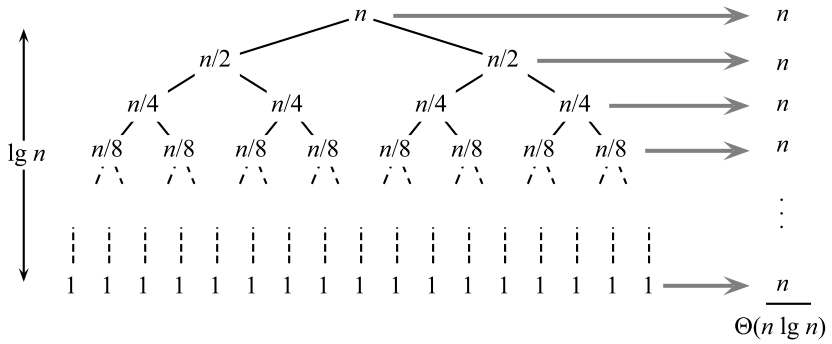


Figura 8.3 Arborele de recurenţă pentru QUICKSORT când procedura PARTIŢIE produce întotdeauna părţi egale (cazul cel mai favorabil). Timpul de execuţie rezultat este $\Theta(n \lg n)$.

a cărei soluţie este $T(n) = \Theta(n \lg n)$ (după cazul 2 al teoremei 4.1). Deci partiţionarea cea mai bună produce un algoritm de sortare mult mai rapid. În figura 8.3 se ilustrează arborele de recursivitate pentru acest cel mai favorabil caz.

Partiţionarea echilibrată

Analiza din secţiunea 8.4 va arăta că timpul mediu de execuţie a algoritmului de sortare rapidă este mult mai apropiat de timpul cel mai bun decât de timpul cel mai rău. Pentru a înţelege de ce este aşa, ar trebui să studiem efectul partiţionării echilibrate asupra formulei recursive care descrie timpul de execuţie.

Să presupunem că procedura de partiţionare produce întotdeauna o împărţire în proporţie de 9 la 1, care la prima vedere pare a fi o partiţionare dezechilibrată. În acest caz, formula recursivă pentru timpul de execuţie al algoritmului de sortare rapidă este:

$$T(n) = T(9n/10) + T(n/10) + n$$

unde, pentru simplificare, în loc de $\Theta(n)$ s-a pus n. Arborele de recurenţă corespunzător se găseşte în figura 8.4. Să observăm că la fiecare nivel al arborelui costul este n până când la adâncimea $\log_{10} n = \Theta(\lg n)$ se atinge o condiţie iniţială. În continuare, la celelalte niveluri, costul nu depăşeşte valoarea n. Apelul recursiv se termină la adâncimea $\log_{10/9} n = \Theta(\lg n)$. Costul total al algoritmului de sortare rapidă este deci $\Theta(n \lg n)$. Ca urmare, cu o partiţionare în proporţie de 9 la 1 la fiecare nivel al partiţionării (care intuitiv pare a fi total dezechilibrată), algoritmul de sortare rapidă are un timp de execuţie de $\Theta(n \lg n)$ – asimptotic acelaşi ca în cazul partiţionării în două părţi egale. De fapt, timpul de execuţie va fi $O(n \lg n)$ şi în cazul partiţionării într-o proporţie de 99 la 1. La orice partiţionare într-o proporţie *constantă*, adâncimea arborelui de recursivitate este $\Theta(\lg n)$ şi costul, la fiecare nivel, este $O(n)$. Deci timpul de execuţie este $\Theta(n \lg n)$ la orice partiţionare într-o proporţie constantă.

Intuirea comportării medii

Pentru a avea o idee clară asupra comportării medii a algoritmului de sortare rapidă, trebuie să facem presupuneri asupra frecvenţei anumitor intrări. Cea mai evidentă presupunere este că

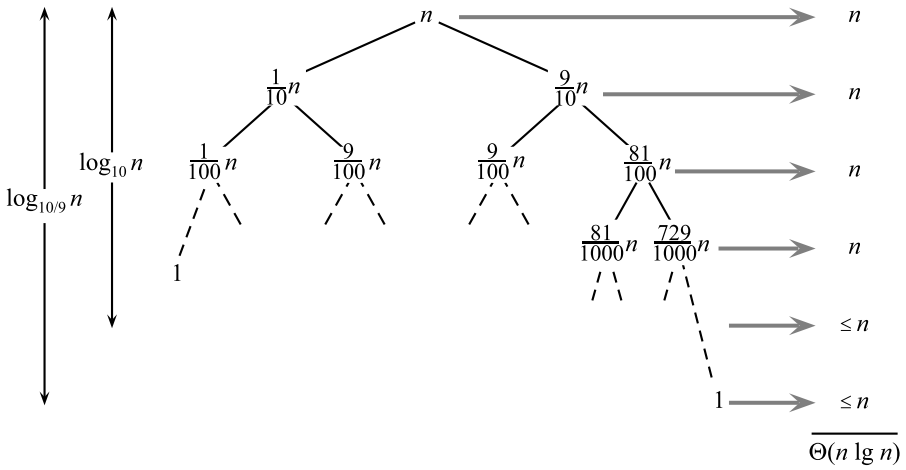


Figura 8.4 Arborele de recurenţă pentru QUICKSORT, când procedura PARTIŢIE produce întotdeauna părţi în proporţie de 9 la 1, rezultând un timp de execuţie de $\Theta(n \lg n)$.

toate permutările elementelor de intrare sunt la fel de probabile. Vom discuta această presupunere în secţiunea următoare, aici vom exploata doar câteva variante.

În situaţia în care algoritmul de sortare rapidă lucrează pe o intrare aleatoare, probabil că nu va partiţiona la fel la fiecare nivel, cum am presupus în discuţiile anterioare. Este de aşteptat ca unele partiţionări să fie echilibrate, altele nu. De exemplu, exerciţiul 8.2-5 cere să se demonstreze că procedura PARTIŢIE produce în 80% din cazuri o partiţionare mai echilibrată decât proporţia de 9 la 1, şi, numai în 20% din cazuri, una mai puţin echilibrată.

În cazul mediu, procedura PARTIŢIE produce un amestec de partiţionări "bune" şi "rele". Într-un arbore de recurenţă pentru cazul mediu al procedurii PARTIŢIE, partiţionările bune şi rele sunt distribuite aleator. Să presupunem, totuşi, pentru simplificare, că partiţionările bune şi rele alternează pe niveluri, şi că partiţionările bune corespund celui mai bun caz, iar cele rele celui mai defavorabilcaz. În figura 8.5 sunt prezentate partiţionările la două niveluri consecutive în arborele de recursivitate. Costul partiţionării la rădăcina arborelui este n, iar vectorii obţinuţi sunt de dimensiune $n-1$ şi 1: cazul cel mai defavorabil. La nivelul următor, vectorul de $n-1$ elemente se împarte în doi vectori de $(n-1)/2$ elemente fiecare, potrivit cazului celui mai bun. Să presupunem că pentru un vector de dimensiune 1 (un element) costul este 1.

Combinarea unei partiţionări rele şi a uneia bune produce trei vectori de dimensiune 1, $(n-1)/2$ şi respectiv $(n-1)/2$, cu un cost total de $2n-1 = \Theta(n)$. Evident, această situaţie nu este mai rea decât cea prezentată în figura 8.5(b), adică cea cu un singur nivel, care produce un vector de $(n-1)/2+1$ elemente şi unul de $(n-1)/2$ elemente, cu un cost total de $n = \Theta(n)$. Totuşi, situaţia din urmă este aproape echilibrată, cu siguranţă mult mai bună decât proporţia 9 la 1. Intuitiv, o partiţionare defavorabilă de un cost $\Theta(n)$ poate fi absorbită de una bună tot de un cost $\Theta(n)$, şi partiţionarea rezultată este favorabilă. Astfel timpul de execuţie al algoritmului de sortare rapidă, când partiţionările bune şi rele alternează, este acelaşi ca în cazul partiţionărilor bune: tot $O(n \lg n)$, doar constanta din notaţia O este mai mare. O analiză riguroasă a cazului mediu se va face în secţiunea 8.4.2.

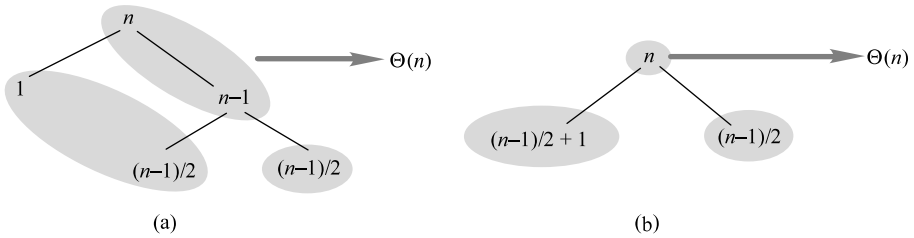


Figura 8.5 **(a)** Două niveluri ale arborelui de recurentă pentru algoritmul de sortare rapidă. Partiţionarea la nivelul rădăcinii consumă n unităţi de timp şi produce o partiţionare "proastă": doi vectori de dimensiune 1 şi $n-1$. Partiţionarea unui subşir de $n-1$ elemente necesită $n-1$ unităţi de timp şi este o partiţionare "bună": produce două subşiruri de $(n-1)/2$ elemente fiecare. **(b)** Un singur nivel al arborelui de recurenţă care este mai rău decât nivelurile combinate de la (a), totuşi foarte bine echilibrat.

Exerciţii

8.2-1 Demonstraţi că timpul de execuţie al algoritmului QUICKSORT, în cazul unui vector A cu toate elementele egale între ele, este $\Theta(n \lg n)$.

8.2-2 Demonstraţi că, în cazul unui vector A având n elemente distincte, ordonate descrescător, timpul de execuţie al algoritmului QUICKSORT este $\Theta(n^2)$.

8.2-3 Băncile, de obicei, înregistrează tranzacţiile pe un cont după data tranzacţiilor, dar multora dintre clienţi le place să primească extrasele de cont, ordonate după numărul cecurilor, deoarece ei primesc cecurile grupate de la comercianţi. Problema convertirii ordonării după data tranzacţiei în ordonare după numărul de cec este problema sortării unui vector aproape ordonat. Să se argumenteze de ce, în acest caz, algoritmul SORTEAZĂ-PRIN-INSERŢIE este mai bun decât algoritmul QUICKSORT.

8.2-4 Să presupunem că partiţionarea la fiecare nivel, în cadrul algoritmului de sortare rapidă, se face într-o proporţie de $1-\alpha$ la α, unde $0 < \alpha \leq 1/2$ şi α este constantă. Demonstraţi că, în arborele de recurenţă, adâncimea minimă a unui nod terminal este aproximativ $-\lg n / \lg \alpha$, pe când adâncimea maximă este aproximativ $-\lg n / \lg(1-\alpha)$. (Nu se va ţine cont de rotunjiri.)

8.2-5 $\star$ Argumentaţi afirmaţia că, pentru orice constantă $0 < \alpha \leq 1/2$, probabilitatea ca pentru o intrare aleatoare procedura PARTIŢIE să producă o partiţionare mai echilibrată decât proporţia $1-\alpha$ la α, este aproximativ $1-2\alpha$. Pentru ce valoare a lui α partiţionarea mai echilibrată are şanse egale cu cea mai puţin echilibrată?

8.3. Variantele aleatoare ale sortării rapide

În analiza comportării medii a algoritmului de sortare rapidă, am presupus că toate permutările elementelor de intrare sunt la fel de probabile. Mulţi consideră că, dacă această presupunere este adevărată, algoritmul de sortare rapidă, este cea mai bună soluţie pentru a sorta vectori de dimensiuni mari. Totuşi, nu ne putem aştepta ca, în practică aceasta să se întâmple întotdeauna

aşa (vezi exerciţiul 8.2-3). În această secţiune, vom introduce noţiunea de algoritm aleator şi vom prezenta două variante aleatoare ale algoritmului de sortare rapidă, care fac inutilă presupunerea că toate permutările elementelor de intrare sunt la fel de probabile.

O alternativă, la *a presupune* o distribuţie de date de intrare, este de *a impune* o astfel de distribuţie. De exemplu, să presupenem că, înainte să sorteze vectorul de intrare, algoritmul de sortare rapidă permută aleator elementele pentru a asigura astfel proprietatea că fiecare permutare are aceeaşi probabilitate. (Exerciţiul 8.3-4 cere scrierea unui algoritm care permută aleator elementele unui vector de dimensiune n în $O(n)$ unităţi de timp.) Această modificare nu îmbunătăţeşte timpul de execuţie a algoritmului în cazul cel mai defavorabil, dar asigură ca timpul de execuţie să nu depindă de ordinea elementelor de intrare.

Un algoritm se numeşte ***aleator*** dacă comportarea lui depinde nu numai de valorile de intrare, ci şi de valorile produse de un ***generator de numere aleatoare***. Vom presupune că dispunem de un generator de numere aleatoare numit RANDOM. Un apel al procedurii RANDOM(a, b) produce un număr întreg între a şi b inclusiv. Fiecare număr întreg din acest interval $[a, b]$ este la fel de probabil. De exemplu, RANDOM(0,1) produce 0 şi 1 cu aceeaşi probabilitate 1/2. Fiecare întreg generat de RANDOM este independent de valorile generate anterior. Ne-am putea imagina că RANDOM este un zar cu $(b - a + 1)$ feţe. (Majoritatea mediilor de programare posedă un ***generator de numere pseudoaleatoare***, un algoritm determinist care produce numere care "par" a fi aleatoare.)

Această variantă a algoritmului de sortare rapidă (ca de altfel mulţi alţi algoritmi aleatori) are interesanta proprietate că, practic, *pentru nici o intrare nu are comportarea cea mai defavorabilă*. Din contră, cazul cel mai defavorabil depinde de generatorul de numere aleatoare. Chiar dacă dorim, nu reuşim să generăm un vector de intrare prost, deoarece permutarea aleatoare face ca ordinea datelor de intrare să fie irelevantă. Algoritmul aleator poate avea o comportare proastă numai dacă generatorul de numere aleatoare produce o intrare nefericită. În exerciţiul 13.4-4 se arată că pentru aproape toate permutările elementelor de intrare, algoritmul de sortare rapidă are o comportare similară cu cazul mediu şi această comportare este apropiată de cea mai defavorabilă doar pentru *foarte* puţine permutări.

O strategie aleatoare este de obicei utilă când există multe moduri în care un algoritm se poate executa, dar este dificil să determinăm un mod care să fie cu siguranţă bun. Dacă multe din aceste alternative sunt bune, alegerea pur şi simplu a uneia, la întâmplare, poate reprezenta o strategie bună. Deseori, în timpul execuţiei sale, un algoritm trebuie să ia multe decizii. Dacă beneficiile unei decizii bune sunt mult mai mari decât costurile unor decizii rele, o selecţie aleatoare de decizii bune şi rele poate determina un algoritm eficient. În secţiunea 8.2 am arătat că o combinaţie de partiţionări bune şi rele asigură un timp de execuţie bun pentru algoritmul de sortare rapidă, şi acest fapt sugerează că şi algoritmul aleator va avea o comportare bună.

Prin modificarea procedurii PARTIŢIE, se poate obţine o variantă nouă a algoritmului de sortare rapidă care foloseşte această strategie de alegere aleatoare. La fiecare pas al sortării rapide, înainte de partiţionarea vectorului, interschimbăm elementul $A[p]$ cu un element ales aleator din vectorul $A[p..r]$. Această modificare asigură ca elementul pivot $x = A[p]$ să fie, cu aceeaşi probabilitate, orice element dintre cele $r - p + 1$ elemente ale vectorului. Astfel, partiţionarea vectorului de intrare poate fi, în medie, rezonabil de echilibrată. Algoritmul aleator, bazat pe permutarea elementelor de intrare, are şi el o comportare medie bună, dar ceva mai dificil de studiat decât această versiune.

Modificările asupra procedurilor PARTIŢIE şi QUICKSORT sunt minore. În noua procedură de partiţionare inserăm pur şi simplu interschimbarea celor două elemente înainte de partiţionare:

PARTIŢIE-ALEATOARE(A, p, r)
1: $i \leftarrow$ RANDOM(p, r)
2: interschimbă $A[p] \leftrightarrow A[i]$
3: **returnează** PARTIŢIE (A, p, r)

Noul algoritm de sortare rapidă foloseşte în loc de PARTIŢIE procedura PARTIŢIE-ALEATOARE:

QUICKSORT-ALEATOR(A, p, r)
1: **dacă** $p < r$ **atunci**
2: $q \leftarrow$ PARTIŢIE-ALEATOARE(A, p, r)
3: QUICKSORT-ALEATOR(A, p, q)
4: QUICKSORT-ALEATOR$(A, q + 1, r)$

Analiza algoritmului se va face în secţiunea următoare.

Exerciţii

8.3-1 De ce, în loc să studiem comportarea cea mai defavorabilă a unui algoritm aleator, studiem comportarea medie?

8.3-2 De câte ori se apelează generatorul de numere aleatoare RANDOM în timpul execuţiei procedurii QUICKSORT-ALEATOR în cazul cel mai defavorabil? Cum se schimbă răspunsul în cazul cel mai favorabil?

8.3-3 $\star$ Descrieţi o implementare a algoritmului RANDOM(a, b), care foloseşte numai aruncări de monede perfecte. Care va fi timpul mediu de execuţie a procedurii?

8.3-4 $\star$ Scrieţi un algoritm aleator care rulează în $\Theta(n)$ unităţi de timp, are ca intrare un vector $A[1..n]$ şi produce la ieşire o permutare a elementelor de intrare.

8.4. Analiza algoritmului de sortare rapidă

Rezultatul studiului intuitiv din secţiunea 8.2 ne face să credem că algoritmul de sortare rapidă are o comportare bună. În această secţiune vom face o analiză mai riguroasă. Începem cu analiza cazului celui mai defavorabil, atât pentru algoritmul QUICKSORT cât şi pentru algoritmul QUICKSORT-ALEATOR, şi terminăm cu analiza cazului mediu pentru algoritmul QUICKSORT-ALEATOR.

8.4.1. Analiza celui mai defavorabil caz

Am văzut, în secţiunea 8.2, că împărţirea cea mai rea la fiecare nivel ne conduce la timpul de execuţie al algoritmului $\Theta(n^2)$, care, intuitiv, reprezintă comportarea în cazul cel mai defavorabil. În continuare vom demonstra această afirmaţie.

Folosind metoda substituţiei (vezi secţiunea 4.1), se poate demonstra că timpul de execuţie al algoritmului QUICKSORT este $O(n^2)$. Fie $T(n)$ timpul de execuţie al algoritmului de sortare

rapidă, în cazul cel mai defavorabil, pentru o intrare de dimensiunea n. Avem următoarea formulă de recurenţă

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n), \tag{8.1}$$

unde parametrul q parcurge valorile între 1 şi $n-1$, deoarece procedura PARTIŢIE produce două subşiruri, având fiecare, cel puţin un element. Presupunem că $T(n) \le cn^2$, pentru o constantă c. Înlocuind în formula (8.1), obţinem următoarele:

$$T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) = c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n).$$

Expresia $q^2 + (n-q)^2$ atinge valoarea sa maximă în domeniul de valori $1 \le q \le n-1$ la una din extreme, deoarece derivata a doua, după q a expresiei, este pozitivă (vezi exerciţiul 8.4-2). Deci, avem

$$\max_{1 \le q \le n-1} (q^2 + (n-q)^2) \le 1^2 + (n-1)^2 = n^2 - 2(n-1).$$

Continuând majorarea lui $T(n)$, obţinem:

$$T(n) \le cn^2 - 2c(n-1) + \Theta(n) \le cn^2,$$

deoarece constanta c poate fi aleasă suficient de mare, astfel încât termenul $2c(n-1)$ să domine termenul $\Theta(n)$. Rezultă că timpul de execuţie (cel mai defavorabil) al algoritmului de sortare rapidă este $\Theta(n^2)$.

8.4.2. Analiza cazului mediu

Am explicat deja, intuitiv, de ce timpul mediu de execuţie al algoritmului QUICKSORT-ALEATOR este $\Theta(n \lg n)$: dacă procedura PARTIŢIE-ALEATOARE împarte vectorul, la fiecare nivel, în aceeaşi proporţie, atunci adâncimea arborelui de recurenţă este $\Theta(\lg n)$, şi timpul de execuţie la fiecare nivel este $\Theta(n)$. Pentru a putea analiza mai exact timpul mediu de execuţie al algoritmului QUICKSORT-ALEATOR, va trebui să înţelegem esenţa procedurii de partiţionare. În continuare, vom putea găsi o formulă de recurenţă pentru timpul mediu de execuţie, necesar pentru sortarea unui vector de dimensiune n. Ca parte a procesului de rezolvare a recurenţei, vom găsi margini tari pentru o sumă interesantă.

Analiza partiţionării

La început, facem câteva observaţii asupra procedurii PARTIŢIE. Când în linia 3 a procedurii PARTIŢIE-ALEATOARE este apelată procedura PARTIŢIE, elementul $A[p]$ este deja interschimbat cu un element ales aleator din vectorul $A[p..r]$. Pentru simplificare, să presupunem că toate elementele de intrare sunt distincte. Dacă elementele de intrare nu sunt distincte, timpul mediu de execuţie al algoritmului rămâne tot $O(n \lg n)$, dar acest caz necesită o analiză ceva mai complexă.

Prima noastră observaţie este că valoarea q, returnată de procedura PARTIŢIE, depinde numai de rangul elmentului $x = A[p]$ în vectorul $A[p..r]$. (***Rangul*** unui element într-o mulţime este egal cu numărul elementelor care sunt mai mici sau egale cu el.) Dacă notăm cu $n = r - p + 1$ numărul elementelor vectorului $A[p..r]$, şi interschimbăm elementul $x = A[p]$ cu un element ales

aleator din vectorul $A[p..r]$, probabilitatea ca rang$(x) = i$ (unde $i = 1, 2, \ldots, n$) este egală cu $1/n$.

Vom calcula, în continuare, probabilitatea diferitelor rezultate ale procedurii de partiţionare. Dacă rang$(x) = 1$, atunci, în procedura PARTIŢIE, la prima execuţie a ciclului **cât timp** din liniile 4–14, variabilele i şi j vor primi aceeaşi valoare p. Deci, când procedura returnează valoarea $q = j$, în primul subşir, singurul element va fi $A[p]$. Probabilitatea acestui caz este $1/n$, deoarece aceasta este probabiltatea ca rang(x) să fie egal cu 1.

Dacă rang$(x) \geq 2$, atunci, există cel puţin un element care este mai mic decât $x = A[p]$. În consecinţă, la prima trecere prin ciclul **cât timp** variabila i va avea valoarea egală cu p, iar j nu va atinge valoarea p. Deci, prin interschimbarea a două elemente între ele, elementul $A[p]$ ajunge în al doilea subşir. Când procedura PARTIŢIE se termină, fiecare dintre cele rang$(x) - 1$ elemente, care se află în primul subşir, vor fi mai mici sau egale cu x. Deci, când rang$(x) \geq 2$, probabilitatea ca în primul subşir să fie i elemente (pentru orice $i = 1, 2, \ldots, n-1$) este egală cu $1/n$.

Combinând cele două cazuri, putem trage concluzia că dimensiunea $q-p+1$, a primului subşir, este egală cu 1, cu probabilitatea $2/n$, iar această dimensiune este egală cu i ($i = 2, 3, \ldots, n-1$), cu probabilitatea $1/n$.

O relaţie de recurenţă pentru cazul mediu

Vom da în continuare o formulă de recurenţă pentru timpul mediu de execuţie al algoritmului QUICKSORT-ALEATOR. Fie $T(n)$ timpul mediu necesar ordonării unui vector având n elemente. Un vector de 1 element poate fi ordonat cu procedura QUICKSORT-ALEATOR în timp constant, deci $T(1) = \Theta(1)$. Procedura QUICKSORT-ALEATOR partiţionează vectorul $A[1..n]$ de dimensiune n în $\Theta(n)$ unităţi de timp. Procedura PARTIŢIE returnează un indice q, iar QUICKSORT-ALEATOR este apelat recursiv pentru un vector de q elemente şi pentru unul cu $n-q$ elemente. Deci timpul mediu pentru ordonarea unui vector, având n elemente, se poate exprima astfel:

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n). \tag{8.2}$$

Distribuţia valorii q este aproape uniformă, exceptând cazul $q = 1$ care este de două ori mai probabil decât celelalte valori, cum am văzut anterior. Folosind valorile $T(1) = \Theta(n)$ şi $T(n-1) = O(n^2)$ (din analiza celui mai defavorabil caz), se poate scrie:

$$\frac{1}{n}(T(1) + T(n-1)) = \frac{1}{n}(\Theta(1) + O(n^2)) = O(n),$$

şi astfel termenul $\Theta(n)$, în formula (8.2), poate absorbi expresia $\frac{1}{n}(T(1) + T(n-1))$. Astfel, formula de recurenţă (8.2) poate fi scrisă sub forma:

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n). \tag{8.3}$$

Se poate observa că, pentru $k = 1, 2, \ldots, n-1$, fiecare termen $T(k)$ al sumei apare de două ori, o dată ca $T(q)$, iar altă dată ca $T(n-q)$. Reducând aceşti termeni, formula finală va fi:

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1}T(k) + \Theta(n). \tag{8.4}$$

Rezolvarea ecuaţiei de recurenţă

Vom putea rezolva ecuaţia de recurenţă (8.4) folosind metoda substituţiei. Să presupunem că $T(n) \leq an \lg n + b$, pentru un $a > 0$ şi $b > 0$. Valorile a şi b pot fi alese destul de mari, astfel încât $an \lg n + b$ să fie mai mare decât $T(1)$. Pentru $n > 1$ avem:

$$\begin{aligned} T(n) &= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \leq \frac{2}{n}\sum_{k=1}^{n-1}(ak \lg k + b) + \Theta(n) = \\ &= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n). \end{aligned}$$

Vom arăta, mai târziu, că ultima sumă poate fi majorată astfel:

$$\sum_{k=1}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2. \tag{8.5}$$

Folosind această majorare, se obţine:

$$\begin{aligned} T(n) &\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \leq \\ &\leq an \lg n - \frac{a}{4}n + 2b + \Theta(n) = an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \leq an \lg n + b, \end{aligned}$$

deoarece valoarea lui a poate fi aleasă astfel încât $\frac{a}{4}n$ să domine expresia $\Theta(n) + b$. Se poate deci trage concluzia că timpul mediu de execuţie a algoritmului de sortare rapidă este $O(n \lg n)$.

Margini strânse pentru însumarea cu chei

Trebuie să mai demonstrăm marginea (8.5) a sumei $\sum_{k=1}^{n-1} k \lg k$.

Deoarece fiecare termen al sumei este cel mult $n \lg n$, se poate scrie:

$$\sum_{k=1}^{n-1} k \lg k \leq n^2 \lg n,$$

formulă care reprezintă o estimare destul de precisă, abstracţie făcând de o constantă, dar nu este suficient de puternică pentru a obţine $T(n) = O(n \lg n)$ ca soluţie a recurenţei. Pentru a obţine soluţia de mai sus, avem nevoie de o margine de forma $\frac{1}{2}n^2 \lg n - \Omega(n^2)$.

Această margine se poate obţine folosind metoda de împărţire a sumei în două, ca în secţiunea 3.2. Vom obţine:

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k.$$

Termenul $\lg k$ din prima sumă a membrului drept se poate majora prin $\lg(n/2) = \lg n - 1$, iar termenul $\lg k$ din cea de a doua sumă prin $\lg n$. Astfel se obţine

$$\sum_{k=1}^{n-1} k \lg k \leq (\lg n - 1)\sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k = \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \leq$$

$$\leq \frac{1}{2}n(n-1)\lg n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2} \leq \frac{1}{2}n^2\lg n - \frac{1}{8}n^2$$

dacă $n \geq 2$. Deci am obţinut chiar formula (8.5).

Exerciţii

8.4-1 Demonstraţi că timpul de execuţie al algoritmului de sortare rapidă, în cazul cel mai favorabil este $\Omega(n \lg n)$.

8.4-2 Demonstraţi că expresia $q^2 + (n-q)^2$, unde $q = 1, 2, \ldots, n-1$, atinge maximul pentru $q = 1$ sau $q = n-1$.

8.4-3 Demonstraţi că timpul mediu de execuţie al lui QUICKSORT-ALEATOR este $\Omega(n \lg n)$.

8.4-4 În practică, timpul de execuţie al algoritmului de sortare rapidă se poate îmbunătăţi considerabil, ţinând cont de execuţia mai rapidă a algoritmului de sortare prin inserţie pentru date de intrare "aproape" ordonate. Când algoritmul de sortare rapidă este apelat pentru un subşir având mai puţin de k elemente, impunem ca acesta să returneze vectorul fără a-l sorta. După revenirea din apelul iniţial, să se execute algoritmul de sortare prin inserţie pentru întregul vector (care este "aproape" ordonat). Argumentaţi afirmaţia că acest algoritm de sortare are un timp mediu de execuţie $O(nk + n\lg(n/k))$. Cum trebuie aleasă valoarea lui k, teoretic şi practic?

8.4-5 $\star$ Demonstraţi următoarea identitate:

$$\int x \ln x dx = \frac{1}{2}x^2 \ln x - \frac{1}{4}x^2,$$

şi, folosind metoda de calculul aproximativ al integralei, daţi o margine mai bună decât cea din (8.5) pentru $\sum_{k=1}^{n-1} k \lg k$.

8.4-6 $\star$ Modificaţi procedura PARTIŢIE, alegând aleator trei elemente din vectorul A, iar dintre acestea elementul de mijloc ca valoare pivot. Aproximaţi probabilitatea ca, în cazul cel mai defavorabil, să se obţină o partiţionare în proporţie de α la $(1-\alpha)$, unde $0 < \alpha < 1$.

Probleme

8-1 Corectitudinea partiţiei

Argumentaţi afirmaţia că procedura PARTIŢIE din secţiunea 8.1 este corectă. Demonstraţi următoarele:

- ***a.*** Indicii i şi j nu se referă niciodată la elemente ale lui A din afara intervalului $[p..r]$.
- ***b.*** La terminarea procedurii PARTIŢIE, indicele j nu va fi niciodată egal cu r (astfel, partiţionarea nu este niciodată trivială).
- ***c.*** La terminarea partiţionării fiecare element din $A[p..j]$ este mai mic sau egal cu orice element din $A[j+1..r]$.

8-2 Algoritmul de partiţionare a lui Lomuto
Considerăm următoarea variantă a procedurii PARTIŢIE, dată de N. Lomuto. Vectorul $A[p..r]$ se împarte în vectorii $A[p..i]$ şi $A[i+1..j]$ astfel încât fiecare element din primul vector este mai mic sau egal cu $x = A[r]$ şi fiecare element din cel de al doilea vector este mai mare ca x.

```
PARTIŢIE-LOMUTO(A, p, r)
 1: x ← A[r]
 2: i ← p − 1
 3: pentru j ← p, r execută
 4:    dacă A[j] ≤ x atunci
 5:       i ← i + 1
 6:       interschimbă A[i] ↔ A[j]
 7: dacă i < r atunci
 8:    returnează i
 9: altfel
10:    returnează i − 1
```

a. Argumentaţi că procedura PARTIŢIE-LOMUTO este corectă.

b. Cel mult, de câte ori poate fi mutat un element în procedura PARTIŢIE? Dar în procedura PARTIŢIE-LOMUTO?

c. Argumentaţi că timpul de execuţie al procedurii PARTIŢIE-LOMUTO, ca de altfel şi al procedurii PARTIŢIE, pe un subşir de n elemente, este $\Theta(n)$.

d. Cum afectează, înlocuirea procedurii PARTIŢIE cu procedura PARTIŢIE-LOMUTO, timpul de execuţie al algoritmului de sortare rapidă QUICKSORT când toate datele de intrare sunt egale?

e. Definiţi o procedură PARTIŢIE-LOMUTO-ALEATOARE care interschimbă elementul $A[r]$ cu un element ales aleator din vectorul $A[p..r]$, apoi apelează procedura PARTIŢIE-LOMUTO. Demonstraţi că probabilitatea ca procedura PARTIŢIE-LOMUTO-ALEATOARE să returneze valoarea q, este aceeaşi cu probabilitatea ca procedura PARTIŢIE-ALEATOARE să returneze valoarea $p + r - q$.

8-3 Sortare circulară
Profesorii Howard, Fine şi Howard au propus următorul algoritm "elegant" pentru sortare:

```
SORTEAZĂ-CIRCULAR(A, i, j)
 1: dacă A[i] > A[j] atunci
 2:    interschimbă A[i] ↔ A[j]
 3: dacă i + 1 ≥ j atunci
 4:    revenire
 5: k ← ⌊(j − i + 1)/3⌋                ▷ Trunchiere.
 6: SORTEAZĂ-CIRCULAR(A, i, j − k)     ▷ Primele două-treimi.
 7: SORTEAZĂ-CIRCULAR(A, i + k, j)     ▷ Ultimele două-treimi.
 8: SORTEAZĂ-CIRCULAR(A, i, j − k)     ▷ Primele două-treimi din nou.
```

a. Demonstraţi că SORTEAZĂ-CIRCULAR$(A, 1,$*lungime*$[A])$ ordonează corect vectorul de intrare $A[1..n]$, unde $n =$ *lungime*$[A]$.

b. Determinaţi o formulă de recurenţă, pentru timpul de execuţie al procedurii SORTEAZĂ-CIRCULAR, şi o margine asimptotică tare (folosind notaţia Θ), pentru timpul de execuţie, în cazul cel mai defavorabil.

c. Comparaţi timpul de execuţie al algoritmului SORTEAZĂ-CIRCULAR, în cazul cel mai defavorabil, cu timpii corespunzători ai algoritmilor de sortare prin inserare, sortare prin interclasare, heapsort şi sortare rapidă. Merită profesorii să fie felicitaţi?

8-4 Adâncimea stivei pentru algoritmul de sortare rapidă

Algoritmul QUICKSORT, prezentat în secţiunea 8.1, se autoapelează de două ori. După apelul procedurii PARTIŢIE, este sortat recursiv primul subşir, apoi subşirul al doilea. Al doilea apel recursiv din QUICKSORT nu este neapărat necesar, el poate fi evitat prin folosirea unei structuri de control iterative. Această tehnică, numită ***recursivitate de coadă***, este oferită în mod automat de compilatoarele mai bune. Se consideră următoarea variantă a algoritmului de sortare rapidă, care simulează recursivitatea de coadă:

QUICKSORT$'(A, p, r)$
1: **cât timp** $p < r$ **execută**
2: $\triangleright$ Partiţionarea şi sortarea primului subşir
3: $q \leftarrow$ PARTIŢIE(A, p, r)
4: QUICKSORT$'(A, p, q)$
5: $p \leftarrow q + 1$

a. Arătaţi că algoritmul QUICKSORT$'(A, 1,$*lungime*$[A])$ sortează corect vectorul A.

Compilatoarele, de obicei, execută apelurile recursive, folosind o ***stivă*** care conţine informaţii pertinente, inclusiv valorile parametrilor la fiecare apel. Informaţiile referitoare la cel mai recent apel se găsesc în vârful stivei, pe când informaţiile referitoare la primul apel se păstrează la baza stivei. Când se apelează o procedură, informaţiile sale se depun în vârful stivei (operaţia ***push***), iar la terminarea execuţiei, aceste informaţii se scot din stivă (operaţia ***pop***). Deoarece presupunem că parametrii vectorului se reprezintă cu ajutorul pointerilor, informaţiile necesare fiecărui apel necesită un spaţiu de stivă de $O(1)$ unităţi. ***Adâncimea stivei*** este spaţiul maxim utilizat de stivă, în timpul execuţiei algoritmului.

b. Descrieţi un scenariu în care, pentru un vector de intrare de dimensiune n, adâncimea stivei algoritmului QUICKSORT$'$ este $\Theta(n)$.

c. Modificaţi textul algoritmului QUICKSORT$'$ astfel încât adâncimea stivei, în cazul cel mai defavorabil, să fie $\Theta(\lg n)$. Păstraţi valoarea $O(n \lg n)$ pentru timpul mediu de execuţie a algoritmului.

8-5 Partiţionare mijlocul-din-3

O metodă prin care procedura QUICKSORT-ALEATOR poate fi îmbunătăţită este de a face partiţionarea în jurul unui element x, ales mai atent decât aleator. O metodă obişnuită poate fi metoda ***mijlocul-din-3***, care constă în a alege elementul mijlociu ca mărime, dintre trei elemente

alese la întâmplare din vector. Să considerăm, pentru problema de faţă, că toate elementele vectorului de intrare $A[1..n]$ sunt distincte şi că $n \geq 3$. Să notăm cu $A'[1..n]$ vectorul sortat rezultat. Fie $p_i = \Pr\{x = A'[i]\}$ probabilitatea ca elementul pivot x, ales cu metoda mijlocul-din-3, să fie al i-lea element în vectorul de ieşire.

a. Daţi o formulă exactă pentru p_i în funcţie de n şi i, pentru $i = 2, 3, \ldots, n-1$. (Să observăm că $p_1 = p_n = 0$.)

b. Cu cât creşte probabilitatea alegerii ca pivot a elementului $x = A'[\lfloor (n+1)/2 \rfloor]$, mijlocul vectorului $A[1..n]$ faţă de implementarea originală? Calculaţi limita acestei probabilităţi când $n \to \infty$.

c. Dacă numim partiţionare "bună" acea partiţionare care foloseşte ca pivot elementul $x = A'[i]$, unde $n/3 \leq i \leq 2n/3$, atunci cu cât creşte probabilitatea obţinerii unei împărţiri bune faţă de implementarea originală? (*Indicaţie:* Aproximaţi suma cu o integrală.)

d. Arătaţi că metoda mijlocul-din-3 afectează numai factorul constant din expresia $\Omega(n \lg n)$ a timpului de execuţie al algoritmului de sortare rapidă.

Note bibliografice

Algoritmul de sortare rapidă a fost inventat de Hoare [98]. Sedgewick [174] oferă o prezentare bună a detaliilor de implementare a acestui algoritm. Avantajele algoritmilor aleatori au fost tratate de Rabin [165].

9 Sortare în timp liniar

În capitolele precedente au fost prezentaţi câţiva algoritmi de complexitate $O(n \lg n)$. Această margine superioară este atinsă de algoritmii de sortare prin interclasare şi heapsort în cazul cel mai defavorabil; respectiv pentru quicksort corespunde în medie. Mai mult decât atât, pentru fiecare din aceşti algoritmi putem efectiv obţine câte o secvenţă de n numere de intrare care determină execuţia algoritmului în timpul $\Omega(n \lg n)$.

Aceşti algoritmi au o proprietate interesantă: *ordinea dorită este determinată în exclusivitate pe baza comparaţiilor între elementele de intrare*. Astfel de algoritmi de sortare se numesc ***sortări prin comparaţii***. Toţi algoritmii de sortare prezentaţi până acum sunt de acest tip.

În secţiunea 9.1 vom demonstra că oricare sortare prin comparaţii trebuie să facă, în cel mai defavorabil caz, $\Omega(n \lg n)$ comparaţii pentru a sorta o secvenţă de n elemente. Astfel, algoritmul de sortare prin interclasare şi heapsort sunt asimptotic optimale şi nu există nici o sortare prin comparaţii care să fie mai rapidă decât cu un factor constant.

Secţiunile 9.2, 9.3 şi 9.4 studiază trei algoritmi de sortare – sortare prin numărare, ordonare pe baza cifrelor şi ordonare pe grupe – care se execută în timp liniar. Se subînţelege că aceşti algoritmi folosesc, pentru a determina ordinea de sortare, alte operaţii decât comparaţiile. În consecinţă, lor nu li se aplică marginea inferioară $\Omega(n \lg n)$.

9.1. Margini inferioare pentru sortare

Într-o sortare prin comparaţii folosim numai comparaţiile dintre elemente pentru a câştiga informaţii de ordine despre o secvenţă de intrare $\langle a_1, a_2, ..., a_n \rangle$. Respectiv, date fiind elementele a_i şi a_j, facem testele $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, sau $a_i > a_j$ pentru a determina ordinea lor relativă. Nu putem să inspectăm valorile elementelor sau să obţinem informaţii asupra ordinii lor în nici un alt mod.

În această secţiune vom presupune, fără a pierde din generalitate, că toate elementele de intrare sunt distincte. Dată fiind această presupunere, comparaţiile de tipul $a_i = a_j$ sunt nefolositoare, deci putem să afirmăm că nu sunt făcute comparaţii de acest fel. Studiind comparaţiile $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ şi $a_i < a_j$ observăm că acestea sunt echivalente, deoarece aduc informaţie identică despre ordinea relativă a lui a_i şi a_j. De aceea vom considera că toate comparaţiile au forma $a_i \leq a_j$.

Modelul arborelui de decizie

Sortările prin comparaţii pot fi viziualizate în mod abstract în termenii ***arborilor de decizie***. Un arbore de decizie reprezintă comparaţiile realizate de un algoritm de sortare când acesta operează asupra unor date de intrare având o mărime dată. Controlul, mişcarea datelor şi toate celelalte aspecte ale algoritmului sunt ignorate. Figura 9.1 vizualizează un arbore de decizie corespunzător algoritmului de sortare prin inserţie din secţiunea 1.1 ce operează cu o secvenţă de intrare având trei elemente.

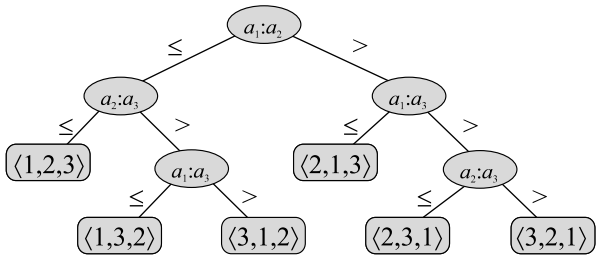


Figura 9.1 Arborele de decizie corespunzător sortării prin inserţie care operează cu trei elemente. Sunt posibile $3! = 6$ permutări între elementele de intrare, deci arborele de decizie trebuie să aibă cel puţin 6 frunze.

Într-un arbore de decizie fiecare nod intern este etichetat prin $a_i : a_j$ pentru i şi j din intervalul $1 \leq i, j \leq n$, unde n este numărul de elemente din secvenţa de intrare. Fiecare frunză este etichetată cu o permutare $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$. (Vezi secţiunea 6.1 pentru noţiuni de bază asupra permutărilor.) Execuţia algoritmului de sortare corespunde trasării unui drum de la rădăcina arborelui de decizie la o frunză. La fiecare nod intern este făcută o comparaţie $a_i \leq a_j$. Subarborele stâng dictează comparaţiile următoare pentru $a_i \leq a_j$, iar subarborele drept dictează comparaţiile următoare pentru $a_i > a_j$. Când ajungem la o frunză, algoritmul de sortare a stabilit ordonarea $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$. Pentru ca algoritmul de sortare să ordoneze adecvat, fiecare dintre cele $n!$ permutări de n elemente trebuie să apară în dreptul unei frunze a arborelui de decizie.

O margine inferioară în cel mai defavorabil caz

Lungimea celui mai lung drum de la rădăcina unui arbore de decizie la oricare dintre frunzele sale reprezintă numărul de comparaţii, în cel mai defavorabil caz, pe care le realizează algoritmul de sortare. În consecinţă, numărul de comparaţii, în cel mai defavorabil caz, pentru o sortare prin comparaţii corespunde înălţimii arborelui de decizie. O margine inferioară a înălţimilor arborilor de decizie este astfel o margine inferioară a timpului de execuţie al oricărui algoritm de sortare prin comparaţii. Următoarea teoremă stabileşte o astfel de margine inferioară.

Teorema 9.1 Orice arbore de decizie care sortează n elemente are înălţimea $\Omega(n \lg n)$.

Demonstraţie. Să considerăm un arbore de decizie având înălţimea h şi care sortează n elemente. Întrucât există $n!$ permutări ale celor n elemente, fiecare permutare reprezentând o ordine distinctă de sortare, arborele trebuie să aibă cel puţin $n!$ frunze. Deoarece un arbore binar având înălţimea h nu are mai mult de 2^h frunze, avem

$$n! \leq 2^h,$$

iar, prin logaritmare, rezultă

$$h \geq \lg(n!),$$

deoarece funcţia lg este monoton crescătoare. Din aproximarea lui Stirling (2.11), avem

$$n! > \left(\frac{n}{e}\right)^n,$$

unde $e = 2.71828...$ este baza logaritmilor naturali; astfel

$$h \geq \lg\left(\frac{n}{e}\right)^n = n\lg n - n\lg e = \Omega(n\lg n).$$

■

Corolarul 9.2 Metoda heapsort şi metoda de sortare prin interclasare sunt metode de sortare prin comparaţii asimptotic optimale.

Demonstraţie. În cazul metodelor de sortare heapsort şi sortarea prin interclasare, marginile superioare $O(n\lg n)$ asupra duratei timpilor de execuţie corespund marginii inferioare $\Omega(n\lg n)$ stabilite de teorema 9.1 pentru cazul cel mai defavorabil. ■

Exerciţii

9.1-1 Care este cea mai mică adâncime posibilă a unei frunze într-un arbore de decizie pentru un algoritm de sortare?

9.1-2 Obţineţi margini asimptotic strânse la $\lg(n!)$ fără a folosi aproximaţia Stirling. În loc de această aproximaţie, evaluaţi suma $\sum_{k=1}^{n}\lg k$ folosind tehnicile prezentate în cadrul secţiunii 3.2.

9.1-3 Arătaţi că nu există nici o sortare prin comparaţii având timpul de execuţie liniar pentru cel puţin jumătate din cele $n!$ intrări de lungime n. Ce se poate spune în cazul unei fracţiuni $1/n$ din datele de intrare de lungime n? Dar despre o fracţiune $1/2^n$?

9.1-4 Profesorul Solomon afirmă că marginea inferioară $\Omega(n\lg n)$ pentru sortarea a n numere nu se aplică în cazul mediului de care dispune, unde după o singură comparaţie $a_i : a_j$, programul se ramifică în trei direcţii, după cum $a_i < a_j, a_i = a_j$ respectiv $a_i > a_j$. Demonstraţi că profesorul nu are dreptate demonstrând că numărul comparaţiilor tridirecţionale cerut de sortarea celor n elemente este tot $\Omega(n\lg n)$.

9.1-5 Demonstraţi că, pentru a interclasa două liste ordonate ce conţin fiecare câte n elemente, sunt necesare $2n - 1$ comparaţii în cazul cel mai defavorabil.

9.1-6 Se consideră o secvenţă de n elemente care trebuie sortate. Datele de intrare sunt formate din n/k subsecvenţe, fiecare conţinând k elemente. Toate elementele dintr-o subsecvenţă dată sunt mai mici decât elementele din subsecvenţa următoare şi mai mari decât elementele din subsecvenţa precedentă. Astfel, pentru sortarea întregii secvenţe de lungime n este suficientă sortarea celor k elemente din fiecare dintre cele n/k subsecvenţe. Găsiţi o margine inferioară $\Omega(n\lg k)$ a numărului de comparaţii necesar pentru rezolvarea acestei variante a problemei de sortare. (*Indicaţie:* combinarea simplă a marginilor inferioare pentru subecvenţele individuale nu este riguroasă.)

9.2. Sortarea prin numărare

Sortarea prin numărare presupune că fiecare dintre cele n elemente ale datelor de intrare este un număr întreg între 1 şi k, pentru un număr întreg k oarecare. Când $k = O(n)$, sortarea se execută în timpul $O(n)$.

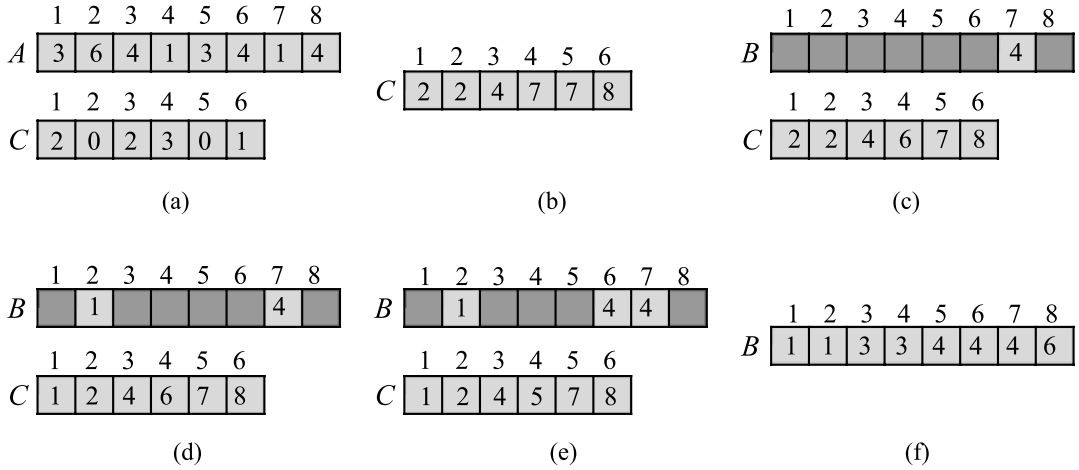


Figura 9.2 Modul de funcţionare al algoritmului SORTARE-PRIN-NUMĂRARE pe un tablou $A\,[1..8]$ de intrare, unde fiecare element din tabloul A este un număr întreg pozitiv nu mai mare decât $k = 6$. **(a)** Tabloul A şi tabloul auxiliar C după liniei 4. **(b)** Tabloul C după executarea liniei 7. **(c)-(e)** Tabloul de ieşire B şi tabloul auxiliar C după unul, două, respectiv trei iteraţii ale buclei din liniile 9–11. Doar elementele haşurate din tabloul B au fost completate. **(f)** Tabloul B sortat, furnizat la ieşire.

Ideea de bază în sortarea prin numărare este de a determina numărul de elemente mai mici decât x, pentru fiecare element x din datele de intrare. Acestă informaţie poate fi utilizată pentru a poziţiona elementul x direct pe poziţia sa din tabloul de ieşire. De exemplu, dacă există 17 elemente mai mici decât x, atunci x va fi pe poziţia 18 în tabloul de ieşire. Această schemă trebuie uşor modificată în situaţia în care există mai multe elemente având aceeaşi valoare, întrucât nu dorim să le aşezăm pe toate în aceeaşi poziţie.

În algoritmul pentru sortarea prin numărare presupunem că datele de intrare formează un tablou $A[1..n]$, şi deci $lungime\,[A] = n$. Sunt necesare alte două tablouri suplimentare, tabloul $B\,[1..n]$, care cuprinde datele de ieşire sortate, şi tabloul $C\,[1..k]$, pentru stocarea temporară în timpul lucrului.

SORTARE-PRIN-NUMĂRARE(A, B, k)

1: **pentru** $i \leftarrow 1, k$ **execută**
2: $\quad C\,[i] \leftarrow 0$
3: **pentru** $j \leftarrow 1, lungime\,[A]$ **execută**
4: $\quad C\,[A\,[j]] \leftarrow C\,[A\,[j]] + 1$
5: ▷ $C[i]$ conţine acum numărul elementelor egale cu i.
6: **pentru** $i \leftarrow 2, k$ **execută**
7: $\quad C\,[i] \leftarrow C\,[i] + C\,[i-1]$
8: ▷ $C[i]$ conţine numărul elementelor mai mici sau egale cu i.
9: **pentru** $j \leftarrow lungime\,[A]\,, 1, -1$ **execută**
10: $\quad B\,[C\,[A\,[j]]] \leftarrow A\,[j]$
11: $\quad C\,[A\,[j]] \leftarrow C\,[A\,[j]] - 1$

Algoritmul de sortare prin numărare este ilustrat în figura 9.2. După iniţializarea din liniile

1–2, în liniile 3–4 se contorizează fiecare element de intrare. Dacă valoarea unui element de intrare este i, se incrementează valoarea lui $C[i]$. Astfel, după liniile 3–4, $C[i]$ păstrează un număr de elemente de intrare egal cu i pentru fiecare număr întreg $i = 1, 2, \ldots, k$. În liniile 6–7 se determină, pentru fiecare $i = 1, 2, \ldots, k$, numărul elementelor de intrare mai mici sau egale decât i; aceasta se realizează prin păstrarea în $C[k]$ a sumei primelor k elemente din tabloul iniţial.

În final, în liniile 9–11, fiecare element $A[j]$ se plasează pe poziţia sa corect determinată din tabloul de ieşire B, astfel încât acesta este ordonat. Dacă toate cele n elemente sunt distincte, la prima execuţie a liniei 9, pentru fiecare $A[j]$, valoarea $C[A[j]]$ este poziţia finală corectă a lui $A[j]$ în tabloul de ieşire, întrucât există $C[A[j]]$ elemente mai mici sau egale cu $A[j]$. Deoarece elementele ar putea să nu fie distincte, decrementăm valoarea $C[A[j]]$ de fiecare dată când plasăm o valoare $A[j]$ în tabloul B; aceasta face ca următorul element de intrare cu o valoare egală cu $A[j]$, dacă există vreunul, să meargă în poziţia imediat dinaintea lui $A[j]$ în tabloul de ieşire.

Cât timp necesită sortarea prin numărare? Bucla **pentru** din liniile 1–2 necesită timpul $O(k)$, bucla **pentru** din liniile 3–4 necesită timpul $O(n)$, bucla **pentru** din liniile 6–7 necesită timpul $O(k)$, iar bucla **pentru** din liniile 9–11 necesită timpul $O(n)$. Deci timpul total este $O(k + n)$. În practică se utilizează sortarea prin numărare când avem $k = O(n)$, caz în care timpul necesar este $O(n)$.

Sortarea prin numărare are un timp mai bun decât marginea inferioară egală cu $\Omega(n \lg n)$ demonstrată în secţiunea 9.1, deoarece nu este o sortare prin comparaţii. De fapt, în cod nu apar comparaţii între elementele de intrare. În schimb, sortarea prin numărare foloseşte valorile elementului ca indice într-un tablou. Marginea inferioară $\Omega(n \lg n)$ nu se aplică atunci când ne îndepărtăm de modelul sortării prin comparaţii.

O proprietate importantă a sortării prin numărare este ***stabilitatea***: numerele cu aceeaşi valoare apar în tabloul de ieşire în aceeaşi ordine în care se găsesc în tabloul de intrare. Adică, legăturile dintre două numere sunt distruse de regula conform căreia oricare număr care apare primul în vectorul de intrare, va apărea primul şi în vectorul de ieşire. Desigur, stabilitatea este importantă numai când datele învecinate sunt deplasate împreună cu elementul în curs de sortare. Vom vedea în secţiunea următoare de ce stabilitatea este importantă.

Exerciţii

9.2-1 Folosind figura 9.2 ca model, ilustraţi modul de operare al algoritmului SORTARE-PRIN-NUMĂRARE pe tabloul $A = \langle 7, 1, 3, 1, 2, 4, 5, 7, 2, 4, 3 \rangle$.

9.2-2 Demonstraţi că SORTARE-PRIN-NUMĂRARE este stabil.

9.2-3 Se presupune că bucla **pentru** din linia 9 a procedurii SORTARE-PRIN-NUMĂRARE este rescrisă sub forma:

pentru $j \leftarrow 1$, $lungime[A]$ **execută**

Arătaţi că algoritmul modificat astfel funcţionează corect. Algoritmul modificat este stabil?

9.2-4 Se presupune că ieşirea algoritmului de sortare este un flux de date, de exemplu, o afişare în mod grafic. Modificaţi algoritmul SORTARE-PRIN-NUMĂRARE pentru a obţine rezultatul în ordinea sortată fără a utiliza vectori suplimentari în afara lui A şi C. (*Indicaţie:* legaţi elementele tabloului A având aceleaşi chei în liste. Unde există un spaţiu "liber" pentru păstrarea pointerilor listei înlănţuite?)

9.2-5 Date fiind n numere întregi din intervalul [1,k], preprocesaţi datele de intrare şi apoi răspundeţi într-un timp $O(1)$ la orice interogare privind numărul total de valori întregi dintre cele n, care se situează în intervalul $[a..b]$. Algoritmul propus ar trebui să folosească un timp de preprocesare $O(n+k)$.

9.3. Ordonare pe baza cifrelor

Ordonarea pe baza cifrelor este algoritmul folosit de către maşinile de sortare a cartelelor, care se mai găsesc la ora actuală numai în muzeele de calculatoare. Cartelele sunt organizate în 80 de coloane, şi fiecare coloană poate fi perforată în 12 poziţii. Sortatorul poate fi "programat" mecanic să examineze o coloană dată a fiecărei cartele dintr-un pachet şi să distribuie fiecare cartela într-una dintre cele 12 cutii, în funcţie de poziţia perforată. Un operator poate apoi să strângă cartelele cutie cu cutie, astfel încât cartelele care au prima poziţie perforată să fie deasupra cartelelor care au a doua poziţie perforată ş.a.m.d.

Pentru cifre zecimale sunt folosite numai 10 poziţii în fiecare coloană. (Celelalte două poziţii sunt folosite pentru codificarea caracterelor nenumerice). Un număr având d cifre ar ocupa un câmp format din d coloane. Întrucât sortatorul de cartele poate analiza numai o singură coloană la un moment dat, problema sortării a n cartele în funcţie de un număr având d cifre necesită un algoritm de sortare.

Intuitiv, am putea dori să sortăm numere în funcţie de *cea mai semnificativă* cifră, să sortăm recursiv fiecare dintre cutiile ce se obţin, şi apoi să combinăm pachetele în ordine. Din nefericire, întrucât cartelele din 9 dintre cele 10 cutii trebuie să fie pastrate pentru a putea sorta fiecare dintre cutii, această procedură generează teancuri intermediare de cartele care trebuie urmărite. (Vezi exerciţiul 9.3-5.)

Ordonarea pe baza cifrelor rezolvă problema sortării cartelelor într-un mod care contrazice intuiţia, sortând întâi în funcţie de *cea mai puţin semnificativă* cifră. Cartelele sunt apoi combinate într-un singur pachet, cele din cutia 0 precedând cartelele din cutia 1, iar acestea din urmă precedând pe cele din cutia 2 şi aşa mai departe. Apoi întregul pachet este sortat din nou în funcţie de a doua cifră cea mai puţin semnificativă şi recombinat apoi într-o manieră asemănătoare. Procesul continuă până când cartelele au fost sortate pe baza tuturor celor d cifre. De remarcat că în acel moment cartelele sunt complet sortate în funcţie de numărul având d cifre. Astfel, pentru sortare sunt necesare numai d treceri prin lista de numere. În figura 9.3 este ilustrat modul de operare al algoritmului de ordonare pe baza cifrelor pe un "pachet" de şapte numere de câte trei cifre.

Este esenţial ca sortarea cifrelor în acest algoritm să fie stabilă. Sortarea realizată de către un sortator de cartele este stabilă, dar operatorul trebuie să fie atent să nu schimbe ordinea cartelelelor pe măsură ce acestea ies dintr-o cutie, chiar dacă toate cartelele dintr-o cutie au aceeaşi cifră în coloana aleasă.

Într-un calculator care funcţionează pe bază de acces secvenţial aleator, ordonarea pe baza cifrelor este uneori utilizată pentru a sorta înregistrările de informaţii care sunt indexate cu chei având câmpuri multiple. De exemplu, am putea dori să sortăm date în funcţie de trei parametri: an, lună, zi. Am putea executa un algoritm de sortare cu o funcţie de comparare care, considerând două date calendaristice, compară anii, şi dacă există o legătură, compară lunile, iar dacă apare din nou o legătură, compară zilele. Alternativ, am putea sorta informaţia de trei ori cu o sortare

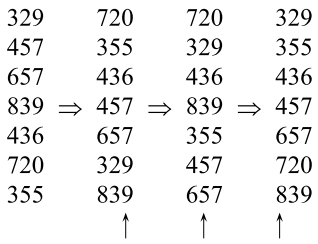


Figura 9.3 Modul de funcţionare al algoritmului de ordonare pe baza cifrelor pe o listă de şapte numere a câte 3 cifre. Prima coloană este intrarea. Celelalte coloane prezintă lista după sortări succesive în funcţie de poziţiile cifrelor în ordinea crescătoare a semnificaţiei. Săgeţile verticale indică poziţia cifrei după care s-a sortat pentru a produce fiecare listă din cea precedentă.

stabilă: prima după zi, următoarea după lună, şi ultima după an.

Pseudocodul pentru algoritmul de ordonare pe baza cifrelor este evident. Următoarea procedură presupune că într-un tablou A având n elemente, fiecare element are d cifre; cifra 1 este cifra cu ordinul cel mai mic, iar cifra d este cifra cu ordinul cel mai mare.

ORDONARE-PE-BAZA-CIFRELOR(A, d)
1: **pentru** $i \leftarrow 1, d$ **execută**
2: folosește o sortare stabilă pentru a sorta tabloul A după cifra i

Corectitudinea algoritmului de ordonare pe baza cifrelor poate fi demonstrată prin inducţie după coloanele care sunt sortate (vezi exerciţiul 9.3-3). Analiza timpului de execuţie depinde de sortarea stabilă folosită ca algoritm intermediar de sortare. Când fiecare cifră este în intervalul $[1, k]$, iar k nu este prea mare, sortarea prin numărare este opţiunea evidentă. Fiecare trecere printr-o mulţime de n numere a câte d cifre se face în timpul $\Theta(n + k)$. Se fac d treceri, astfel încât timpul total necesar pentru algoritmul de ordonare pe baza cifrelor este $\Theta(dn + kd)$. Când d este constant şi $k = O(n)$, algoritmul de ordonare pe baza cifrelor se execută în timp liniar.

Unii informaticieni consideră că numărul biţilor într-un cuvânt calculator este $\Theta(\lg n)$. Pentru exemplificare, să presupunem că $d \lg n$ este numărul de biţi, unde d este o constantă pozitivă. Atunci, dacă fiecare număr care va fi sortat încape într-un cuvânt al calculatorului, îl vom putea trata ca pe un număr având d cifre reprezentat în baza n. De exemplu, să considerăm sortarea a 1 milion de numere având 64 de biţi. Tratând aceste numere ca numere de patru cifre în baza 2^{16}, putem să le sortăm pe baza cifrelor doar prin patru treceri, comparativ cu o sortare clasică prin comparaţii de timp $\Theta(n \lg n)$ care necesită aproximativ $\lg n = 20$ de operaţii pentru fiecare număr sortat. Din păcate, versiunea algoritmului de ordonare pe baza cifrelor care foloseşte sortarea prin numărare ca sortare intermediară stabilă nu sortează pe loc, lucru care se întâmplă în cazul multora din sortările prin comparaţii de timp $\Theta(n \lg n)$. Astfel, dacă se doreşte ca necesarul de memorie să fie mic, atunci este preferabil algoritmul de sortare rapidă.

Exerciţii

9.3-1 Folosind figura 9.3 ca model, ilustraţi modul de fucţionare al algoritmului ORDONARE-PE-BAZA-CIFRELOR pe următoarea listă de cuvinte în limba engleză: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

9.3-2 Care din următorii algoritmi de sortare sunt stabili: sortarea prin inserţie, sortarea prin interclasare, heapsort şi sortarea rapidă? Realizaţi o schemă simplă care să facă stabil orice algoritm de sortare. Cât timp şi spaţiu suplimentar necesită schema realizată?

9.3-3 Demonstraţi prin inducţie că algoritmul de ordonare pe baza cifrelor funcţionează corect. Unde avem nevoie în demonstraţie de presupunerea că sortarea intermediară este stabilă?

9.3-4 Arătaţi cum se sortează n întregi din intervalul $[1, n^2]$ în timp $O(n)$.

9.3-5 $\star$ Determinaţi numărul exact de treceri necesare pentru a sorta numere zecimale având d cifre, în cel mai defavorabil caz, pentru primul algoritm din acest capitol pentru ordonarea cartelelor. De câte teancuri de cartele ar avea nevoie un operator pentru a urmări această sortare?

9.4. Ordonarea pe grupe

Ordonarea pe grupe se execută, în medie, în timp liniar. Ca şi sortarea prin numărare, Ordonarea pe grupe este rapidă pentru că face anumite presupuneri despre datele de intrare. În timp ce sortarea prin numărare presupune că intrarea constă din întregi dintr-un domeniu mic, ordonarea pe grupe presupune că intrarea este generată de un proces aleator care distribuie elementele în mod uniform în intervalul $[0, 1)$. (Vezi secţiunea 6.2 pentru definiţia distribuţiei uniforme.)

Principiul algoritmului de ordonare pe grupe este de a împărţi intervalul $[0, 1)$ în n subintervale egale, numite ***grupe*** (engl. *buckets*) şi apoi să distribuie cele n numere de intrare în aceste grupe. Întrucât datele de intrare sunt uniform distribuite în intervalul $[0, 1)$, nu ne aşteptăm să fie prea multe numere în fiecare grupă. Pentru a obţine rezultatul dorit, sortăm numerele din fiecare grupă, apoi trecem prin fiecare grupă în ordine, listând elementele din fiecare.

Pseudocodul nostru pentru ordonarea pe grupe presupune că datele de intrare formează un tablou A având n elemente şi că fiecare element $A[i]$ satisface relaţia $0 \leq A[i] < 1$. Codul necesită un tablou auxiliar $B[0..n-1]$ de liste înlănţuite (reprezentând grupele) şi presupune că există un mecanism pentru menţinerea acestor liste. (În secţiunea 11.2 se descrie cum se implementează operaţiile de bază pe listele înlănţuite.)

ORDONARE-PE-GRUPE(A)
1: $n \leftarrow lungime[A]$
2: **pentru** $i \leftarrow 1, n$ **execută**
3: inserează $A[i]$ în lista $B\left[\lfloor nA\left[i\right]\rfloor\right]$
4: **pentru** $i \leftarrow 0, n-1$ **execută**
5: sortează lista $B[i]$ folosind sortarea prin inserţie
6: concatenează în ordine listele $B\left[0\right], B\left[1\right], ..., B\left[n-1\right]$

Figura 9.4 ilustrează modul de funcţionare al algoritmului de ordonare pe grupe pe un tablou de intrare cu 10 numere.

Pentru a demonstra că acest algoritm funcţionează corect, se consideră două elemente $A\left[i\right]$ şi $A\left[j\right]$. Dacă aceste elemente sunt distribuie în aceeaşi grupă, ele apar în ordinea relativă adecvată în secvenţa de ieşire, deoarece grupa lor este sortată de sortarea prin inserţie. Să presupunem, totuşi, că ele sunt distribuite în grupe diferite. Fie aceste grupe $B[i']$ şi respectiv $B[j']$, şi să

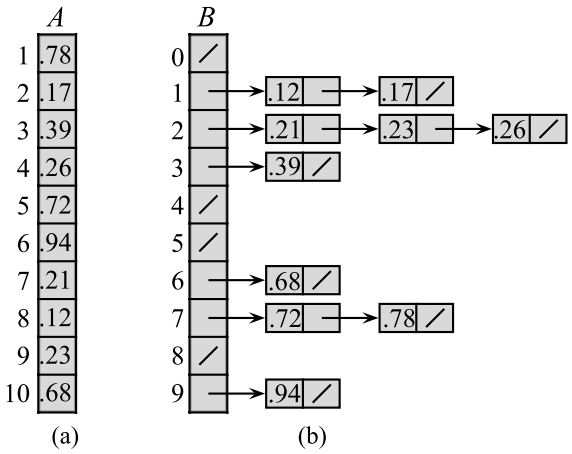


Figura 9.4 Funcţionarea algoritmului ORDONARE-PE-GRUPE. **(a)** Tabloul de intrare $A[1..10]$. **(b)** Tabloul $B[0..9]$ al listelor (reprezentând grupele) sortate după linia a cincea a algoritmului. Grupa i cuprinde valorile din intervalul $[i/10, (i+1)/10)$. Ieşirea sortată constă dintr-o concatenare în ordine a listelor $B[0]$, $B[1], ..., B[9]$.

presupunem, fără a pierde din generalitate, că $i' < j'$. Când listele lui B sunt concatenate în linia 6, elementele grupei $B[i']$ apar înaintea elementelor lui $B[j']$, şi astfel $A[i]$ precede $A[j]$ în secvenţa de ieşire. Deci trebuie să arătăm că $A[i] \leq A[j]$. Presupunând contrariul, avem

$$i' = \lfloor nA[i] \rfloor \geq \lfloor nA[j] \rfloor = j',$$

ceea ce este o contradicţie, întrucât $i' < j'$. Aşadar, ordonarea pe grupe funcţionează corect.

Pentru a analiza timpul de execuţie, să observăm mai întâi că toate liniile, cu excepţia liniei 5, necesită, în cel mai defavorabil caz, timpul $O(n)$. Timpul total pentru a examina în linia 5 toate grupele este $O(n)$ şi, astfel, singura parte interesantă a analizei este timpul consumat de sortările prin inserţie din linia 5.

Pentru a analiza costul sortărilor prin inserţie, fie n_i o variabilă aleatoare desemnând numărul de elemente din grupa $B[i]$. Întrucât sortarea prin inserţie se execută în timp pătratic (vezi secţiunea 1.2), timpul necesar sortării elementelor în grupele $B[i]$ este $E\left[O\left(n_i^2\right)\right] = O\left(E\left[n_i^2\right]\right)$. În consecinţă, timpul mediu total necesar sortării tuturor elementelor în toate grupele va fi

$$\sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right) = O\left(\sum_{i=0}^{n-1} E\left[n_i^2\right]\right). \tag{9.1}$$

Pentru a evalua această sumă trebuie să determinăm distribuţia pentru fiecare variabilă aleatoare n_i. Avem n elemente în n grupe. Probabilitatea ca un element dat să intre într-o grupă $B[i]$ este $1/n$, întrucât fiecare grupă este responsabilă pentru $1/n$ elemente din intervalul $[0, 1)$. Astfel, situaţia este asemănătoare cu exemplul aruncatului mingii din secţiunea (6.6.2): avem n mingi (elemente) şi n cutii (reprezentând grupele), şi fiecare minge este aruncată independent, cu probabilitatea $p = 1/n$ de a intra într-o anume grupă. Astfel, probabilitatea ca $n_i = k$ urmează o distribuţie binomială $b(k; n, p)$ care are media $E[n_i] = np = 1$ şi varianta $Var[n_i] = np(1-p) =$

$1 - 1/n$. Pentru fiecare variabilă aleatoare X, ecuaţia (6.30) este echivalentă cu

$$E\left[n_i^2\right] = Var\left[n_i\right] + E^2\left[n_i\right] = 1 - \frac{1}{n} + 1^2 = 2 - \frac{1}{n} = \Theta\left(1\right).$$

Utilizând acest salt în ecuaţia (9.1) concluzionăm că timpul mediu pentru sortarea prin inserţie este $O(n)$. Astfel, întregul algoritm de ordonare pe grupe se execută în timp mediu liniar.

9.4.1. Exerciţii

9.4-1 Utilizând ca model figura 9.4, ilustraţi modul de funcţionare al algoritmului ORDONARE-PE-GRUPE pe tabloul $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42\rangle$.

9.4-2 Care este timpul de execuţie în cazul cel mai defavorabil pentru algoritmul de ordonare pe grupe? Ce modificare simplă a algoritmului păstrează timpul mediu liniar de execuţie şi face ca timpul de execuţie în cazul cel mai defavorabil să fie $O(n \lg n)$?

9.4-3 ⋆ Se dau n puncte în cercul unitate, $p_i = (x_i, y_i)$, astfel încât $0 < x_i^2 + y_i^2 \leq 1$ pentru $i = 1, 2, ..., n$. Presupuneţi că punctele sunt uniform distribuite, adică probabilitatea de a găsi un punct în orice regiune a cercului este proporţională cu aria acelei regiuni. Scrieţi un algoritm în timpul mediu $\Theta\left(n\right)$ care să sorteze cele n puncte în funcţie de distanţele faţă de origine $d_i = \sqrt{x_i^2 + y_i^2}$. (*Indicaţie:* proiectaţi dimensiunile grupelor din ORDONAREA-PE-GRUPE pentru a reflecta distribuţia uniformă a punctelor în cercul unitate.)

9.4-4 ⋆ O ***funcţie de distribuţie a probabilităţii*** $P(x)$ pentru variabila aleatoare X este definită de $P(x) = \Pr\{X \leq x\}$. Consideraţi o listă de n numere cu o funcţie de distribuţie de probabilitate continuă P care este calculabilă în timpul $O(1)$. Arătaţi cum se sortează numerele în timp mediu liniar.

Probleme

9-1 Marginile inferioare în cazul mediu la sortarea prin comparaţii
În această problemă demonstrăm o margine inferioară $\Omega\left(n \lg n\right)$ a timpului mediu de execuţie pentru orice sortare prin comparaţii deterministă sau aleatoare efectuată pe n date de intrare. Începem prin examinarea unei sortări prin comparaţii deterministe A având arborele de decizie T_A. Presupunem că fiecare permutare a intrărilor lui A apare cu o probabilitate constantă.

a. Să presupunem că fiecare frunză a lui T_A este etichetată cu probabilitatea de a fi atinsă în cazul unor date de intrare aleatoare. Demonstraţi că exact $n!$ frunze sunt etichetate cu $1/n!$ şi că restul sunt etichetate cu 0.

b. Fie $D(T)$ lungimea drumului extern a arborelui T; adică, $D(T)$ este suma adâncimilor tuturor frunzelor lui T. Fie T un arbore având $k > 1$ frunze, şi fie LT şi RT subarborii stâng şi drept ai lui T. Arătaţi că $D(T) = D(LT) + D(RT) + k$.

c. Fie $d(k)$ valoarea minimă a lui $D(T)$ pe toţi arborii de decizie T având $k > 1$ frunze. Arătaţi că $d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\}$. (*Indicaţie:* Consideraţi un arbore T având k frunze care atinge minimul. Fie i numărul de frunze din LT şi $k - i$ numărul de frunze din RT.)

d. Demonstraţi că pentru o valoare dată a lui $k > 1$ şi i din domeniul $1 \leq i \leq k-1$, funcţia $i \lg i + (k-i) \lg(k-i)$ este minimă pentru $i = k/2$. Arătaţi că $d(k) = \Omega(k \lg k)$.

e. Demonstraţi că $D(T_A) = \Omega(n! \lg(n!))$ pentru T_A, şi concluzionaţi că timpul mediu pentru sortarea a n elemente este $\Omega(n \lg n)$.

Acum să considerăm o sortare *aleatoare* prin comparaţii B. Putem extinde modelul arborelui de decizie pentru a gestiona caracterul aleator prin încorporarea a două feluri de noduri: noduri de comparaţie şi noduri de "randomizare". Un astfel de nod "modelează" o selectare aleatoare de forma ALEATOR$(1, r)$, făcută de către algoritmul B; nodul are r descendenţi, fiecare dintre ei având aceeaşi probabilitate de a fi selectat în timpul execuţiei algoritmului.

f. Arătaţi că, pentru orice sortare aleatoare prin comparaţii B, există o sortare deterministă prin comparaţii A care nu face în medie mai multe comparaţii decât B.

9-2 Sortarea pe loc în timp liniar

a. Să presupunem că avem de sortat un tablou cu n articole şi cheia fiecărui articol are valoarea 0 sau 1. Realizaţi un algoritm simplu, de timp liniar, care să sorteze pe loc cele n articole. Pe lângă memoria necesară reprezentării tabloului, puteţi folosi un spaţiu suplimentar de memorie având dimensiune constantă.

b. Poate sortarea realizată la punctul (a) să fie utilizată pentru o ordonare pe baza cifrelor de timp $O(bn)$ a n articole având chei de b-biţi? Explicaţi cum sau de ce nu.

c. Să presupunem că cele n înregistrări au chei în intervalul $[1, k]$. Arătaţi cum se poate modifica sortarea prin numărare astfel încât înregistrările să poată fi sortate pe loc în timpul $O(n+k)$. Puteţi folosi un spaţiu suplimentar de memorie de ordinul $O(k)$, pe lângă tabloul de intrare. (*Indicaţie:* Cum aţi realiza acest lucru pentru $k = 3$?)

Note bibliografice

Modelul arborelui de decizie pentru studiul sortărilor prin comparaţii a fost introdus de către Ford şi Johnson [72]. Tratatul cuprinzător al lui Knuth asupra sortării [123] acoperă multe variaţii pe problema sortării, inclusiv marginea inferioară teoretică asupra complexităţii sortării prezentate aici. Marginile inferioare pentru sortare folosind generalizările modelului arborelui de decizie au fost studiate exhaustiv de către Ben-Or [23].

Knuth îi atribuie lui H.H. Seward inventarea sortării prin numărare în 1954 şi a ideii de combinare a sortării prin numărare cu ordonare pe baza cifrelor. Ordonarea pe baza cifrelor după cea mai puţin semnificativă cifră pare să fie un algoritm popular, larg utilizat de către operatorii maşinilor mecanice de sortare a cartelelor. Conform lui Knuth, prima referinţă publicată a metodei este un document din 1929 aparţinând lui L.J. Comrie, ce descrie echipamentul pentru cartele perforate. Ordonarea pe grupe a fost utilizată începând cu 1956, când ideea de bază a fost propusă de E.J. Isaac şi R.C. Singleton.

10 Mediane şi statistici de ordine

A i-a ***statistică de ordine*** a unei mulţimi de n elemente este al i-lea cel mai mic element. De exemplu, ***minimul*** unei mulţimi de elemente este prima statistică de ordine ($i = 1$), iar ***maximul*** este a n-a statistică de ordine ($i = n$). Intuitiv, o ***mediană*** este "punctul de la jumătatea drumului" unei mulţimi. Când n este impar, mediana este unică, şi apare pe poziţia $i = (n+1)/2$. Când n este par, există două mediane, care apar pe poziţiile $i = n/2$ şi $i = n/2+1$. Astfel, indiferent de paritatea lui n, medianele apar pe poziţiile $i = \lfloor (n+1/2) \rfloor$ (***mediana inferioară***) şi $i = \lceil (n+1/2) \rceil$ (***mediana superioară***). Pentru simplificare, vom folosi de acum înainte termenul "mediană" cu sensul de mediană inferioară.

Acest capitol tratează problema selectării celei de-a i-a statistici de ordine dintr-o mulţime de n numere distincte. Presupunem că mulţimea conţine numere distincte, deşi teoretic tot ce facem se extinde la situaţia în care o mulţime conţine valori nu neapărat distincte. ***Problema selecţiei*** poate fi enunţată riguros după cum urmează:

Date de intrare: O mulţime A de n numere (distincte) şi un număr i, având proprietatea $1 \leq i \leq n$.

Date de ieşire: Elementul $x \in A$, care este mai mare decât exact $i-1$ alte elemente ale lui A.

Problema selecţiei poate fi rezolvată în timp $O(n \lg n)$, întrucât numerele se pot sorta utilizând heapsort sau sortarea prin interclasare şi apoi, indexând, pur şi simplu al i-lea element din tabloul de ieşire. Există totuşi şi algoritmi mai rapizi.

În secţiunea 10.1 se examinează problema selectării minimului şi maximului unei mulţimi de elemente. Mai interesantă este problema selecţiei generale, detaliată în următoarele două secţiuni. Secţiunea 10.2 analizează un algoritm practic care atinge o margine $O(n)$ în timp mediu de execuţie. Secţiunea 10.3 conţine un algoritm de interes mai mult teoretic care atinge timpul de execuţie $O(n)$ în cel mai defavorabil caz.

10.1. Minim şi maxim

Câte comparaţii sunt necesare pentru a determina minimul unei mulţimi de n elemente? Putem obţine uşor o margine superioară de $n-1$ comparaţii: se examinează pe rând fiecare element al mulţimii şi se reţine cel mai mic element vizitat până la momentul curent. În procedura MINIM presupunem că mulţimea constă din tabloul A, unde $lungime\,[A] = n$.

```
MINIM(A)
1: min ← A[1]
2: pentru i ← 2, lungime[A] execută
3:    dacă min > A[i] atunci
4:       min ← A[i]
5: returnează min
```

Determinarea maximului poate fi realizată la fel de bine şi prin $n-1$ comparaţii.

Este acesta cel mai bun algoritm? Da, întrucât putem obţine o margine inferioară pentru $n-1$ comparaţii în problema determinării minimului. Să ne gândim la un algoritm care determină minimul în cazul unui campionat. Fiecare comparaţie reprezintă un meci al campionatului pe care îl câştigă cel mai mic dintre două elemente. Observaţia cheie este că fiecare element, cu excepţia învingătorului, trebuie să piardă cel puţin un meci. Deci, pentru a determina minimul, sunt necesare $n-1$ comparaţii, iar algoritmul MINIM este optim în privinţa numărului de comparaţii realizate.

Un punct interesant al analizei este determinarea numărului mediu de execuţii ale liniei 4. Problema 6-2 vă cere să demonstraţi că această medie este $\Theta(\lg n)$.

Minim şi maxim simultane

În unele aplicaţii trebuie găsite simultan minimul şi maximul unei mulţimi de n elemente. De exemplu, un program grafic poate avea nevoie să scaleze o mulţime de date (x, y) pentru a le afişa pe un ecran dreptunghiular sau pe un alt dispozitiv de ieşire. Pentru a realiza acest lucru, programul trebuie, mai întâi, să determine minimul şi maximul fiecărei coordonate.

Nu este prea dificil de proiectat un algoritm care să poată găsi atât minimul cât şi maximul a n elemente folosind numărul asimptotic optimal de comparaţii $\Omega(n)$. Se determină minimul şi maximul independent, folosind $n-1$ comparaţii pentru fiecare, deci un total de $2n-2$ comparaţii.

De fapt, sunt suficiente numai $3\lceil n/2 \rceil - 2$ comparaţii pentru a determina atât minimul cât şi maximul. Prima pereche necesită numai o comparaţie pentru a stabili valorile iniţiale pentru minimul şi maximul curente, ceea ce justifică termenul -2. În acest scop, păstrăm minimul şi maximul găsite pe măsură ce se lucrează. În loc să prelucrăm fiecare element de intrare comparându-l cu minimul şi maximul curente cu un cost de două comparaţii per element, vom prelucra perechi de elemente. Comparăm perechile de elemente ale datelor de intrare, mai întâi *între ele*, apoi comparăm elementul mai mic cu minimul curent şi elementul mai mare cu maximul curent, cu un cost de trei comparaţii pentru fiecare două elemente.

Exerciţii

10.1-1 Arătaţi că al doilea cel mai mic dintre n elemente poate fi determinat, în cel mai defavorabil caz, prin $n + \lceil \lg n \rceil - 2$ comparaţii. (*Indicaţie:* Găsiţi şi cel mai mic element.)

10.1-2 $\star$ Arătaţi că, în cel mai defavorabil caz, sunt necesare $\lceil 3n/2 \rceil - 2$ comparaţii pentru a determina maximul şi minimul a n elemente. (*Indicaţie:* Studiaţi câte numere pot fi potenţial maximul sau minimul, şi cercetati cum afectează o comparaţie aceste calcule.)

10.2. Selecţia în timp mediu liniar

Problema generală a selecţiei pare mai dificilă decât simpla problemă a găsirii unui minim, şi în plus, surprinzător, timpul asimptotic de execuţie pentru ambele probleme este acelaşi: $\Theta(n)$. În această secţiune vom prezenta un algoritm divide şi stăpâneşte pentru problema selecţiei. Algoritmul SELECŢIE-ALEATOARE este modelat pe baza algoritmului de sortare rapidă din capitolul 8. Ca şi la sortarea rapidă, ideea este de a partiţiona recursiv tabloul de intrare. Spre deosebire de sortarea rapidă, care prelucrează recursiv ambele componente ale partiţiei,

SELECŢIE-ALEATOARE lucrează numai cu o componentă. Această diferenţă se evidenţiază la analiză: în timp ce sortarea rapidă are un timp mediu de execuţie de $\Theta(n \lg n)$, timpul mediu al algoritmului SELECŢIE-ALEATOARE este $\Theta(n)$.

SELECŢIE-ALEATOARE foloseşte procedura PARTIŢIE-ALEATOARE, prezentată în secţiunea 8.3. Astfel, ca şi QUICKSORT-ALEATOR, este un algoritm aleator, întrucât comportamentul său este parţial determinat de datele de ieşire ale unui generator de numere aleatoare. Codul următor pentru SELECŢIE-ALEATOARE returnează al i-lea cel mai mic element al tabloului $A[p..r]$.

```
SELECŢIE-ALEATOARE(A, p, r, i)
1: dacă p = r atunci
2:    returnează A[p]
3: q ← PARTIŢIE-ALEATOARE(A, p, r)
4: k ← q − p + 1
5: dacă i ≤ k atunci
6:    returnează SELECŢIE-ALEATOARE(A, p, q, i)
7: altfel
8:    returnează SELECŢIE-ALEATOARE(A, q + 1, r, i − k)
```

După execuţia procedurii PARTIŢIE-ALEATOARE din linia 3 a algoritmului, tabloul $A[p..r]$ este partiţionat în două subtablouri nevide $A[p..q]$ şi $A[q+1..r]$ astfel încât fiecare element al lui $A[p..q]$ este mai mic decât fiecare element al lui $A[q+1..r]$. Linia 4 a algoritmului calculează numărul k de elemente al subtabloului $A[p..q]$. Algoritmul determină în care dintre subtablourile $A[p..q]$ şi $A[q+1..r]$ se situează al i-lea cel mai mic element. Dacă $i \leq k$, atunci elementul dorit se situează în partea inferioară a partiţiei, şi este selectat recursiv din subtabloul accesat în linia 6. Dacă $i > k$, elementul dorit se situează în partea superioară a partiţiei. Întrucât cunoaştem deja k valori mai mici decât al i-lea cel mai mic element al lui $A[p..q]$ – adică elementele lui $A[p..r]$ – elementul căutat este al $i-k$-lea cel mai mic element al lui $A[q+1..r]$, care este găsit recursiv în linia 8.

Timpul de execuţie, în cazul cel mai defavorabil, pentru SELECŢIE-ALEATOARE este $\Theta(n^2)$, chiar şi pentru găsirea minimului, pentru că am putea fi extrem de nenorocoşi şi să partiţionăm în jurul celui mai mare element rămas. Algoritmul lucrează bine în cazul mediu, deşi, deoarece este aleator, nu există date de intrare particulare care să provoace comportamentul celui mai defavorabil caz.

Putem obţine o margine superioară $T(n)$ pe timpul mediu necesar algoritmului SELECŢIE-ALEATOARE pe un tablou de intrare de n elemente, după cum urmează. Am văzut în secţiunea 8.4 că algoritmul PARTIŢIE-ALEATOARE produce o partiţie a cărei componentă inferioară are un singur element cu probabilitatea $2/n$ şi i elemente cu probabilitatea $1/n$ pentru $i = 2, 3, ..., n-1$. Presupunând că $T(n)$ este monoton crescător, în cazul cel mai defavorabil SELECŢIE-ALEATOARE este întotdeauna nenorocos în sensul că cel de-al i-lea element este localizat pe zona cea mai mare a partiţiei. Astfel, obţinem recurenţa

$$\begin{aligned} T(n) &\leq \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n) \\ &\leq \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n) = \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n). \end{aligned}$$

A doua linie rezultă din prima întrucât $\max(1, n-1) = n-1$ şi

$$\max(k, n-k) = \begin{cases} k & \text{dacă } k \geq \lceil n/2 \rceil, \\ n-k & \text{dacă } k < \lceil n/2 \rceil. \end{cases}$$

Dacă n este impar, fiecare termen $T(\lceil n/2 \rceil), T(\lceil n/2 \rceil + 1), \ldots, T(n-1)$ apare de două ori în sumă, şi chiar dacă n este par, fiecare termen $T(\lceil n/2 \rceil + 1)$, $T(\lceil n/2 \rceil + 2), ..., T(n-1)$ apare de două ori, iar termenul $T(\lceil n/2 \rceil)$ apare o singură dată. În oricare dintre cele două cazuri, suma de pe prima linie este mărginită superior de suma de pe a doua linie. În cazul cel mai defavorabil, $T(n-1) = O(n^2)$, şi astfel termenul $\frac{1}{n}T(n-1)$ poate fi incorporat de către termenul $O(n)$.

Demonstrăm recurenţa prin substituţie. Presupunem că $T(n) \leq cn$ pentru unele constante c care satisfac condiţiile iniţiale ale recurenţei. Folosind această ipoteză de inducţie, avem

$$\begin{aligned} T(n) &\leq \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} ck + O(n) = \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k\right) + O(n) \\ &= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\left\lceil \frac{n}{2} \right\rceil - 1\right)\left\lceil \frac{n}{2} \right\rceil\right) + O(n) \\ &\leq c(n-1) - \frac{c}{n}\left(\frac{n}{2} - 1\right)\left(\frac{n}{2}\right) + O(n) = c\left(\frac{3}{4}n - \frac{1}{2}\right) + O(n) \leq cn, \end{aligned}$$

întrucât putem alege un c suficient de mare pentru ca $c(n/4 + 1/2)$ să domine termenul $O(n)$.

Astfel, orice statistică de ordine şi în special mediana, poate fi determinată într-un timp mediu liniar.

Exerciţii

10.2-1 Scrieţi versiunea iterativă a algoritmului SELECŢIE-ALEATOARE.

10.2-2 Să presupunem că folosim SELECŢIE-ALEATOARE pentru a selecta elementul minim al unui tablou $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1 \rangle$. Descrieţi o secvenţă de partiţii care se generează, în cazul cel mai defavorabil, de către SELECŢIE-ALEATOARE.

10.2-3 Amintim că dacă există elemente egale atunci procedura PARTIŢIE-ALEATOARE partiţionează subtabloul $A[p..r]$ în două subtablouri nevide $A[p..q]$ şi $A[q+1..r]$, astfel încât fiecare element din $A[p..q]$ este mai mic *sau egal* decât fiecare element din $A[q+1..r]$. Dacă există elemente egale, mai funcţionează corect procedura SELECŢIE-ALEATOARE?

10.3. Selecţia în timp liniar în cazul cel mai defavorabil

Să examinăm acum un algoritm de selecţie al cărui timp de execuţie pentru cazul cel mai defavorabil este $O(n)$. La fel ca şi SELECŢIE-ALEATOARE, algoritmul SELECŢIE găseşte elementul dorit partiţionând recursiv un tablou de intrare. Ideea algoritmului este de a *garanta* o partiţionare bună a tabloului. SELECŢIE utilizează algoritmul determinist PARTIŢIE din algoritmul de sortare rapidă (vezi secţiunea 8.1), modificat astfel încât să preia ca parametru de intrare elementul în jurul căruia este efectuată partajarea.

Algoritmul SELECŢIE determină al i-lea cel mai mic element al unui tablou de intrare de n elemente, efectuând următorii paşi. (Dacă $n = 1$, atunci SELECŢIE returnează singura dată de intrare a ei, ca fiind a i-a cea mai mică valoare.)

1. Se împart cele n elemente ale tabloului de intrare în $\lfloor n/5 \rfloor$ grupe de câte 5 elemente fiecare şi cel mult un grup să conţină restul de n mod 5 elemente.
2. Se determină mediana fiecăruia din cele $\lceil n/5 \rceil$ grupe cu sortarea prin inserţie, ordonând elementele fiecărui grup (care are cel mult 5 elemente) apoi se aleg medianele din listele sortate, corespunzătoare grupelor. (Dacă grupul are un număr par de elemente se va considera cea mai mare dintre cele două mediane.)
3. Se utilizează recursiv SELECŢIE pentru a găsi mediana x din cele $\lceil n/5 \rceil$ mediane găsite la pasul 2.
4. Se partiţionează tabloul de intrare în jurul medianei medianelor x, utilizând o versiune modificată a algoritmului PARTIŢIE. Fie k numărul de elemente de pe latura inferioară a partiţiei, astfel încât să fie $n - k$ elemente pe latura superioară.
5. Se utilizează recursiv SELECŢIE pentru a găsi al i-lea cel mai mic element de pe latura inferioară dacă $i \leq k$, sau al $(i - k)$-lea cel mai mic element de pe latura superioară dacă $i > k$.

Pentru a analiza timpul de execuţie a lui SELECŢIE, determinăm mai întâi o margine inferioară a numărului de elemente care sunt mai mari decât elementul de partiţionare x. Figura 10.1 ajută la vizualizarea acestei contabilizări. Cel puţin jumătate dintre medianele găsite la pasul 2 sunt mai mari sau egale cu mediana medianelor, x. Astfel, cel puţin jumătate din cele $\lceil n/5 \rceil$ grupe au câte 3 elemente mai mari decât x, cu excepţia singurului grup care are mai puţin de 5 elemente dacă n nu este divizibil cu 5, şi a singurului grup care îl conţine pe x. Eliminând aceste două grupuri, rezultă că numărul elementelor mai mari decât x este cel puţin

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6.$$

Similar, numărul elementelor mai mici decât x este cel puţin $3n/10 - 6$. Astfel, în cel mai defavorabil caz, SELECŢIE este apelat recursiv la pasul 5 pentru cel mult $7n/10 + 6$ elemente.

Putem construi o recurenţă pentru timpul de execuţie $T(n)$, în cazul cel mai defavorabil, al algoritmului SELECŢIE. Paşii 1, 2 şi 4 necesită timpul $O(n)$. (Pasul 2 constă din $O(n)$ apeluri pentru sortarea prin inserţie pe mulţimi de mărimea $O(1)$.) Pasul 3 necesită timpul $T(\lceil n/5 \rceil)$, iar pasul 5 necesită cel mult $T(7n/10 + 6)$, presupunând că T este monoton crescător. De reţinut că $7n/10 + 6 < n$ pentru $n > 20$ şi că orice date de intrare de 80 sau mai puţine elemente necesită timpul $O(1)$. Obţinem deci recurenţa

$$T(n) \leq \begin{cases} \Theta(1) & \text{dacă } n \leq 80, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{dacă } n > 80. \end{cases}$$

Demonstrăm prin substituţie că timpul de execuţie este liniar. Presupunem că $T(n) \leq cn$ pentru o anume constantă c şi toţi $n \leq 80$. Substituind această ipoteză inductivă în partea dreaptă a recurenţei obţinem

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + O(n) \leq cn/5 + c + 7cn/10 + 6c + O(n) \\ &\leq 9cn/10 + 7c + O(n) \leq cn, \end{aligned}$$

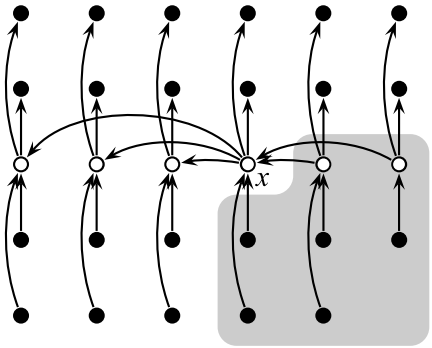


Figura 10.1 Analiza algoritmului SELECŢIE. Cele n elemente sunt reprezentate prin mici cercuri, şi fiecare grup ocupă o coloană. Medianele grupurilor sunt colorate în alb, iar mediana medianelor x este etichetată. Săgeţile sunt trasate de la elementele mai mari spre cele mai mici; se poate vedea că cele 3 elemente din dreapta lui x din fiecare grup de câte 5 sunt mai mari decât x, şi că cele 3 elemente din stânga lui x din fiecare grup de câte 5 elemente sunt mai mici decât x. Elementele mai mari decât x sunt situate pe un fond mai închis.

întrucât putem să alegem un c suficient de mare astfel încât $c(n/10 - 7)$ să fie mai mare decât funcţia descrisă de termenul $O(n)$ pentru orice $n > 80$. Timpul de execuţie $T(n)$ al algoritmului SELECŢIE, în cazul cel mai defavorabil, este deci liniar.

La fel ca şi în cazul sortării prin comparaţii (vezi secţiunea 9.1), SELECŢIE şi SELECŢIE-ALEATOARE determină informaţia despre ordinea relativă a elementelor numai prin compararea lor. Astfel, timpul liniar nu este un rezultat al unor presupuneri despre datele de intrare, cum a fost cazul algoritmilor de sortare din capitolul 9. Sortarea necesită timpul $\Omega(n \lg n)$ în modelul bazat pe comparaţii, chiar în cazul mediu (vezi problema 9-1), şi astfel metoda sortării şi indexării prezentată în introducerea acestui capitol este asimptotic ineficientă.

Exerciţii

10.3-1 În algoritmul SELECŢIE elementele de intrare sunt împărţite în grupe de câte 5. Va mai funcţiona algoritmul în timp liniar dacă ele sunt împărţite în grupe de câte 7? Argumentaţi că SELECŢIE nu se va executa în timp liniar dacă grupele conţin câte 3 elemente.

10.3-2 Analizaţi procedura SELECŢIE şi arătaţi că numărul elementelor mai mari decât mediana medianelor x şi numărul elementelor mai mici decât x este cel puţin $\lceil n/4 \rceil$ dacă $n \geq 38$.

10.3-3 Arătaţi cum poate fi modificat algoritmul de sortare rapidă astfel încât să se execute în timpul $O(n \lg n)$ în cel mai defavorabil caz.

10.3-4 $\star$ Să presupunem că un algoritm utilizează numai comparaţii pentru a determina al i-lea cel mai mic element dintr-o mulţime de n elemente. Arătaţi că el poate determina şi cele $i-1$ elemente mai mici şi cele $n-i$ elemente mai mari, fără a mai face nici o comparaţie în plus.

10.3-5 Fiind dată o procedură de tip "cutie neagră", care în cazul cel mai defavorabil determină mediana în timp liniar, proiectaţi un algoritm simplu, care să rezolve problema selecţiei pentru orice statistică de ordine în timp liniar.

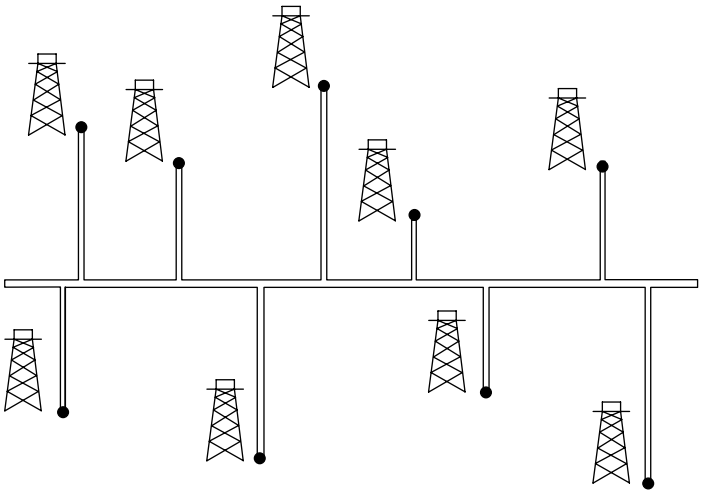


Figura 10.2 Vrem să determinăm poziţia conductei de petrol est-vest care mimimizează lungimea totală a racordurilor nord-sud.

10.3-6 *k*-***cuantilele*** unei mulţimi de n elemente sunt cele $k-1$ statistici de ordine care împart mulţimea sortată în k mulţimi de mărimi egale (cu cel puţin un element). Realizaţi un algoritm în timpul $O(n \lg k)$ care să listeze k-cuantilele unei mulţimi.

10.3-7 Descrieţi un algoritm de timp $O(n)$ care, fiind dată o mulţime S de n numere distincte şi un întreg pozitiv $k \leq n$, determină cele k numere din S care sunt cele mai apropiate de mediana lui S.

10.3-8 Fie $X[1..n]$ şi $Y[1..n]$ două tablouri ce conţin fiecare n numere deja sortate. Scrieţi un algoritm de timp $O(\lg n)$ care găseşte mediana tuturor celor $2n$ elemente din tablourile X şi Y.

10.3-9 Profesorul Olay acordă consultaţii unei companii petroliere, care plănuieşte realizarea unei mari conducte ce trece de la est la vest printr-un câmp de petrol cu n puţuri. La fiecare puţ trebuie conectată direct la conducta principală o conductă de pompare (racord) care să fie de-a lungul celui mai scurt drum (fie nord, fie sud), ca în figura 10.2. Cunoscându-se coordonatele x şi y ale puţurilor, cum trebuie profesorul să localizeze optim conducta principală (cea care minimizează lungimea totală a racordurilor)? Arătaţi că poziţia optimă poate fi determinată în timp liniar.

Probleme

10-1 Cele mai mari i ***numere în ordine sortată***

Fiind dată o mulţime de n numere, dorim să găsim cele mai mari i numere în ordinea sortată, folosind un algoritm bazat pe comparaţii. Descrieţi un algoritm optim care implementează fiecare dintre metodele următoare, şi analizaţi timpii de execuţie ai metodelor în funcţie de n şi i.

a. Sortaţi numerele şi listaţi cele mai mari i numere.

b. Construiţi o coadă de priorităţi din numere şi apelaţi EXTRAGE-MAX de i ori.

c. Utilizaţi un algoritm de statistică de ordine pentru a găsi al i-lea cel mai mare număr, partiţionaţi şi sortaţi cele mai mari i numere.

10-2 Mediana ponderată

Pentru n elemente distincte $x_1, x_2, \ldots, x_n$ cu ponderile pozitive $w_1, w_2, \ldots, w_n$, astfel încât $\sum_{i=1}^{n} w_i = 1$, ***mediana ponderată*** este elementul x_k ce satisface condiţiile

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

şi

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}.$$

(Aceasta este de fapt ***mediana inferioară ponderată***; ***mediana superioară ponderată*** se defineşte similar.)

a. Argumentaţi că mediana elementelor $x_1, x_2, \ldots, x_n$ este mediana ponderată a lui x_i cu ponderile $w_i = 1/n$ pentru $i = 1, 2, \ldots, n$.

b. Arătaţi cum se calculează mediana ponderată a n elemente în timpul $O(n \lg n)$ în cazul cel mai defavorabil, folosind sortarea.

c. Arătaţi cum se calculează mediana ponderată în timpul $\Theta(n)$ în cazul cel mai defavorabil folosind un algoritm de timp mediu liniar cum ar fi, de exemplu, SELECŢIE din secţiunea 10.3.

Problema amplasării oficiului poştal este definită în modul următor. Se consideră n puncte $p_1, p_2, ..., p_n$ cu ponderile asociate $w_1, w_2, ..., w_n$. Se doreşte determinarea unui punct p (nu neapărat unul din punctele de intrare) care să minimizeze suma $\sum_{i=1}^{n} w_i d(p, p_i)$, unde $d(a, b)$ este distanţa dintre punctele a şi b.

d. Argumentaţi că mediana ponderată este cea mai bună soluţie pentru problema amplasării oficiului poştal în cazul unidimensional în care punctele sunt numere reale simple, iar distanţa dintre punctele a şi b este $d(a, b) = |a - b|$.

e. Găsiţi cea mai bună soluţie pentru problema amplasării oficiului poştal în cazul bidimensional, în care punctele sunt perechi de coordonate (x, y) şi distanţa dintre punctele $a = (x_1, y_1)$ şi $b = (x_2, y_2)$ este ***distanţa Manhattan***: $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

10-3 Statistici de ordin mic

Numărul de comparaţii $T(n)$ în cazul cel mai defavorabil folosit de către procedura SELECŢIE pentru a alege statistica de a i-a ordine din n numere s-a demonstrat că satisface relaţia $T(n) = \Theta(n)$, dar constanta ascunsă de notaţia Θ este destul de mare. Când i este mic în raport cu n se poate implementa o procedură diferită care utilizează SELECŢIE ca o subrutină, dar face mai puţine comparaţii în cazul cel mai defavorabil.

a. Descrieţi un algoritm care foloseşte $U_i(n)$ comparaţii pentru a determina al i-lea cel mai mic element dintre n elemente, unde $i \leq n/2$ şi

$$U_i(n) = \begin{cases} T(n) & \text{dacă } n \leq 2i, \\ n/2 + U_i(n/2) + T(2i) & \text{în caz contrar.} \end{cases}$$

(*Indicaţie:* Începeţi cu $\lfloor n/2 \rfloor$ comparaţii pe perechi disjuncte şi apelaţi procedura recursiv pentru mulţimea conţinând elementul mai mic din fiecare pereche.)

b. Arătaţi că $U_i(n) = n + O(T(2i)\lg(n/i))$.

c. Arătaţi că dacă i este constant, atunci $U_i(n) = n + O(\lg n)$.

d. Arătaţi că dacă $i = n/k$ pentru $k \geq 2$, atunci $U_i(n) = n + O(T(2n/k)\lg k)$.

Note bibliografice

Algoritmul de timp liniar în cel mai defavorabil caz pentru determinarea medianei a fost inventat de către Blum, Floyd, Pratt, Rivest şi Tarjan [29]. Versiunea algoritmului pentru un timp mediu rapid se datorează lui Hoare [97]. Floyd şi Rivest [70] au dezvoltat o versiune îmbunătăţită a timpului mediu care partiţionează în jurul unui element selectat recursiv dintr-o mostră mică de elemente.

III Structuri de date

Introducere

Mulţimea ca noţiune fundamentală este la fel de importantă în informatică ca şi în matematică. În timp ce mulţimile matematice sunt nemodificabile, mulţimile manipulate de algoritmi pot creşte, descreşte sau, în restul cazurilor,se pot modifica în timp. Astfel de mulţimi se numesc ***dinamice***. Următoarele cinci capitole prezintă tehnici de bază pentru reprezentarea mulţimilor dinamice finite şi pentru manipularea lor de către calculator.

În funcţie de algoritmi, sunt necesare diferite tipuri de operaţii pentru a fi executate asupra mulţimilor. De exemplu, mulţi algoritmi au nevoie doar de posibilitatea de a insera elemente într-o mulţime, de a şterge elemente dintr-o mulţime sau de a testa apartenenţa la o mulţime. O mulţime dinamică ce suportă aceste operaţii este numită ***dicţionar***. Alţi algoritmi necesită operaţii mai complicate. De exemplu, cozile de prioritate, care sunt introduse în capitolul 7 în contextul structurii de date ansamblu, suportă operaţiile de inserare a unui element şi extragere a celui mai mic element dintr-o mulţime. Deci nu e surprinzător faptul că cea mai bună modalitate de a implementa o mulţime dinamică depinde de operaţiile pe care trebuie să le ofere.

Elementele unei mulţimi dinamice

Într-o implementare tipică a unei mulţimi dinamice, fiecare element este reprezentat de un obiect ale cărui câmpuri pot fi examinate şi manipulate dacă avem un pointer (o referinţă) la acel obiect (capitolul 11 prezintă implementarea obiectelor şi pointerilor în medii de programare care nu le conţin ca tipuri de date de bază). Unele categorii de mulţimi dinamice presupun că un câmp al obiectului este un câmp ***cheie***, de identificare. Dacă toate cheile sunt diferite, mulţimea dinamică poate fi privită ca o mulţime de valori ale cheilor. Obiectul poate conţine ***date adiţionale***, care sunt gestionate în alte câmpuri ale obiectului, dar sunt de fapt neutilizate de către implementarea mulţimii. De asemenea, obiectul poate avea câmpuri ce sunt manipulate de către operaţiile mulţimilor, aceste câmpuri putând conţine date sau referinţe spre alte obiecte din mulţime.

Unele mulţimi dinamice presupun că valorile cheilor sunt alese dintr-o mulţime total ordonată, ca de exemplu mulţimea numerelor reale sau mulţimea tuturor cuvintelor aflate în ordine (uzuală) lexicografică. (O mulţime total ordonată satisface proprietatea de trihotomie, definită la pagina 27). O ordine totală ne permite să definim, de exemplu, cel mai mic element în mulţime sau să vorbim despre următorul element mai mare decât un element dat din mulţime.

Operaţii pe mulţimi dinamice

Operaţiile pe o mulţime dinamică pot fi împărţite în două categorii: ***interogări***, care returnează informaţii despre mulţime sau ***operaţii de modificare***, care modifică mulţimea. În continuare se prezintă o listă a operaţiilor uzuale. Orice aplicaţie specifică necesită în general doar implementarea unora dintre ele.

CAUTĂ(S,k) O interogare care, fiind date o mulţime S şi o valoare cheie k, returnează un pointer x la un element din S astfel încât $cheie[x] = k$ sau NIL dacă nu există un astfel de element în S.

INSEREAZĂ(S,x) O operaţie de modificare care adaugă la mulţimea S elementul referit de x. În mod obişnuit se presupune că toate câmpurile din elementul x necesare implementării mulţimii au fost iniţializate în prealabil.

ŞTERGE(S,x) O operaţie de modificare care, fiind dat un pointer x la un element din mulţimea S, şterge x din S. (Observaţi că această operaţie foloseşte un pointer la un element x şi nu o valoare a cheii).

MINIM(S) O interogare pe o mulţime total ordonată S care returnează elementul din mulţimea S cu cea mai mică valoare a cheii.

MAXIM(S) O interogare pe o mulţime total ordonată S care returnează elementul din mulţimea S cu cea mai mare valoare a cheii.

SUCCESOR(S,x) O interogare care, fiind dat un element x a cărui cheie este dintr-o mulţime S total ordonată, returnează elementul următor mai mare decât x din S sau NIL dacă x este elementul maxim.

PREDECESOR(S,x) O interogare care, fiind dat un element x a cărui cheie este dintr-o mulţime S total ordonată, returnează elementul următor mai mic decât x din S sau NIL dacă x este elementul minim.

Interogările SUCCESOR şi PREDECESOR sunt deseori extinse la mulţimi cu elemente nedistincte. Pentru o mulţime cu n chei, presupunerea obişnuită este că un apel al lui MINIM urmat de $n - 1$ apeluri ale lui SUCCESOR enumeră elementele din mulţime în ordinea sortării.

Timpul necesar pentru a executa o operaţie asupra mulţimilor este, în mod obişnuit, măsurat în termenii mărimii mulţimii, dat ca unul dintre argumente. De exemplu, capitolul 14 descrie o structură de date ce poate realiza oricare dintre operaţiile descrise mai sus pe o mulţime de dimensiune n în timpul $O(\lg n)$.

Sumarul părţii a treia

Capitolele 11–15 descriu diferite structuri de date ce pot fi folosite pentru a implementa mulţimi dinamice; multe dintre acestea vor fi utilizate ulterior la construirea unor algoritmi eficienţi pentru o diversitate de probleme. O altă structură de date importantă – ansamblul – a fost deja introdusă în capitolul 7.

Capitolul 11 prezintă fundamentele lucrului cu structuri de date simple ca: stive, cozi, liste înlănţuite şi arbori cu rădăcină. De asemenea, se arată cum pot fi implementate obiectele sau referinţele în medii de programare care nu le posedă ca primitive. O mare parte din acest material ar trebui să fie familiară tuturor celor care au studiat un curs introductiv în programare.

Capitolul 12 introduce tabele de dispersie (hash), care au operaţii pentru dicţionare: INSEREAZĂ, ŞTERGE, CAUTĂ. În cazul cel mai nefavorabil, dispersia necesită timpul Θ(n) pentru a executa o operaţie CAUTĂ, pe când timpul prevăzut pentru operaţiile de dispersie este $O(1)$. Analiza dispersiei se bazează pe probabilităţi, dar cea mai mare parte a capitolului nu necesită cunoştinţe despre acest subiect.

Arborii de căutare binară, prezentaţi în capitolul 13 au toate operaţiile pe mulţimi dinamice descrise mai sus. În cazul cel mai defavorabil, fiecare operaţie necesită un timp $\Theta(n)$ pe un arbore cu n elemente, dar pe un arbore binar de căutare construit aleator, timpul mediu pentru fiecare

operaţie este $O(\lg n)$. Arborii de căutare binară au rol de bază pentru multe alte structuri de date.

Arborii roşu–negru, o variantă a arborilor de căutare binară, sunt introduşi în capitolul 14. Spre deosebire de arborii de căutare binară obişnuiţi, arborii roşu–negru prezintă garanţia unei execuţii bune: operaţiile necesită un timp $O(\lg n)$ în cazul cel mai defavorabil. Un arbore roşu–negru este un arbore de căutare echilibrat; capitolul 19 prezintă un alt tip de arbori de căutare echilibraţi, numiţi B–arbori. Deşi mecanismul arborilor roşu–negru este într-o oarecare măsură greu de urmărit, majoritatea proprietăţilor lor se poate deduce din capitolul respectiv, fără a studia mecanismul în detaliu. Oricum, parcurgerea codului poate fi instructivă în sine.

În capitolul 15 vom arăta cum se pot îmbogăţi arborii roşu – negru pentru a suporta alte operaţii, în afara celor de bază prezentate mai sus. În primul rând, îi vom îmbogăţi astfel încât să putem păstra dinamic o statistică de ordine pentru o mulţime de chei. Apoi, îi vom îmbogăţi într-o manieră diferită pentru a memora intervale de numere reale.

11 Structuri de date elementare

În acest capitol vom examina reprezentarea mulţimilor dinamice ca structuri de date simple care utilizează referinţe. Deşi multe structuri de date complexe pot fi modelate folosind pointeri, le vom prezenta doar pe cele elementare: stive, cozi, liste înlănţuite şi arbori cu rădăcină. De asemenea, vom prezenta o metodă prin care obiectele şi pointerii pot fi sintetizaţi din tablouri.

11.1. Stive şi cozi

Stivele şi cozile sunt mulţimi dinamice în care elementul care este eliminat din mulţime de către operaţia ŞTERGE este prespecificat. Într-o ***stivă***, elementul şters din mulţime este elementul cel mai recent inserat: stiva implementează principiul ***ultimul sosit, primul servit*** (last-in, first-out) sau ***LIFO***. Similar, într-o ***coadă***, elementul şters este întotdeauna cel care a stat în mulţime cel mai mult (primul introdus): coada implementează principiul ***primul sosit, primul servit*** (first-in, first-out) sau ***FIFO***. Există mai multe modalităţi eficiente de a implementa stivele şi cozile cu ajutorul calculatorului. În această secţiune vom arăta cum se pot folosi tablourile simple pentru a implementa fiecare dintre aceste structuri.

Stive

Operaţia INSEREAZĂ asupra stivelor este deseori denumită PUNE-ÎN-STIVĂ, iar operaţia ŞTERGE, care nu cere nici un argument, este deseori numită SCOATE-DIN-STIVĂ. Aceste nume sunt aluzii la stivele fizice, ca de exemplu un vraf de farfurii. Ordinea în care sunt luate farfuriile din vraf este ordinea inversă în care au fost introduse în vraf, deoarece doar ultima farfurie este accesibilă.

Aşa cum se vede în figura 11.1, o stivă cu cel mult n elemente poate fi implementată printr-un tablou $S[1..n]$. Tabloul are un atribut $vârf[S]$ care este indicele celui mai recent introdus element. Stiva constă din elementele $S[1..vârf[S]]$, unde $S[1]$ este elementul de la baza stivei, iar $S[vârf[S]]$ este elementul din vârful stivei.

Când $vârf[S] = 0$, stiva nu conţine nici un element şi deci este ***vidă***. Se poate testa dacă stiva este vidă prin operaţia de interogare STIVĂ-VIDĂ.

Dacă se încearcă extragerea (se apelează operaţia SCOATE-DIN-STIVĂ) unui element dintr-o stivă vidă, spunem că stiva are ***depăşire inferioară***, care în mod normal este o eroare. Dacă $vârf[S]$ depăşeşte valoarea n, stiva are ***depăşire superioară***. (În implementarea noastră în pseudocod, nu ne vom pune problema depăşirii stivei.)

Fiecare dintre operaţiile stivei pot fi implementate prin doar câteva linii de cod.

STIVĂ-VIDĂ(S)
1: **dacă** $vârf[S] = 0$ **atunci**
2: **returnează** ADEVĂRAT
3: **altfel**
4: **returnează** FALS

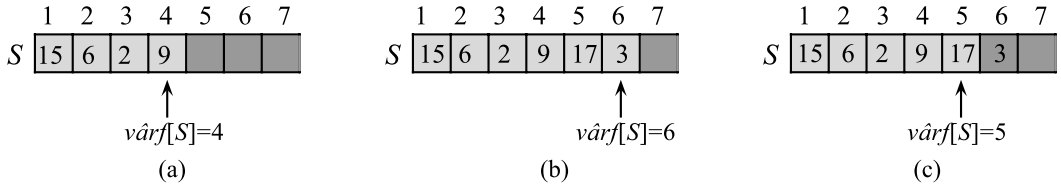


Figura 11.1 Implementarea unei stive S printr-un tablou. Elementele stivei apar doar în poziţiile haşurate folosind o culoare deschisă. **(a)** Stiva S are 4 elemente. Elementul din vârful stivei este 9. **(b)** Stiva S după apelurile PUNE-ÎN-STIVĂ (S,17) şi PUNE-ÎN-STIVĂ(S,3). **(c)** Stiva S după ce apelul SCOATE-DIN-STIVĂ (S) a întors ca rezultat elementul 3, care este cel mai recent introdus. Deşi elementul 3 apare în continuare în tabel, el nu mai este în stivă; elementul din vârful stivei este 17.

PUNE-ÎN-STIVĂ(S, x)
1: *vârf*[S] ← *vârf*[S] + 1
2: *S[vârf*[S]] ← x

SCOATE-DIN-STIVĂ(S)
1: **dacă** STIVĂ-VIDĂ(S) **atunci**
2: **eroare** "depăşire inferioară"
3: **altfel**
4: *vârf*[S] ← *vârf*[S] - 1
5: **returnează** S[*vârf*[S]+1]

Figura 11.1 ilustrează modificările produse în urma apelurilor operaţiilor PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ. Fiecare dintre cele trei operaţii asupra stivelor necesită un timp $O(1)$.

Cozi

Vom numi PUNE-ÎN-COADĂ operaţia INSEREAZĂ aplicată asupra unei cozi, iar operaţia ŞTERGE o vom numi SCOATE-DIN-COADĂ; asemănător operaţiei SCOATE-DIN-STIVĂ aplicată stivei, SCOATE-DIN-COADĂ nu necesită nici un argument. Principiul FIFO, propriu cozii, impune ca aceasta să opereze asemănător cu un rând de oameni ce aşteaptă la un ghişeu. Coada are un ***cap*** şi o ***coadă***. Când un element este pus în coadă, ocupă locul de la sfârşitul cozii, ca şi un nou venit ce îşi ia locul la coada rândului. Elementul scos din coadă este întotdeauna cel din capul cozii, asemănător persoanei din capul rândului care a aşteptat cel mai mult. (Din fericire, nu trebuie să ne îngrijorăm din cauza elementelor care "se bagă în faţă".)

Figura 11.2 ilustrează o modalitate de a implementa o coadă având cel mult $n-1$ elemente, folosind un tablou $Q[1..n]$. Coada are un atribut $cap[Q]$ care conţine indicele capului ei. Atributul $coadă[Q]$ conţine indicele noii locaţii în care se va insera în coadă elementul nou venit. Elementele din coadă se află în locaţiile $cap[Q], cap[Q]+1, ..., coadă[Q]-1$, iar indicii se parcurg circular, în sensul că locaţia 1 urmează imediat după locaţia n. Când $cap[Q] = coadă[Q]$ coada este goală. Iniţial avem $cap[Q] = coadă[Q] = 1$. Când coada este vidă, o încercare de a scoate un element din coadă cauzează o depăşire inferioară în coadă. Când $cap[Q] = coadă[Q] + 1$, coada este "plină" şi o încercare de a pune în coadă cauzează o depăşire superioară în coadă.

În PUNE-ÎN-COADĂ şi SCOATE-DIN-COADĂ, verificarea erorilor de depăşire inferioară a fost omisă. (Exerciţiul 11.1-4 cere scrierea codului ce verifică aceste două condiţii de eroare.)

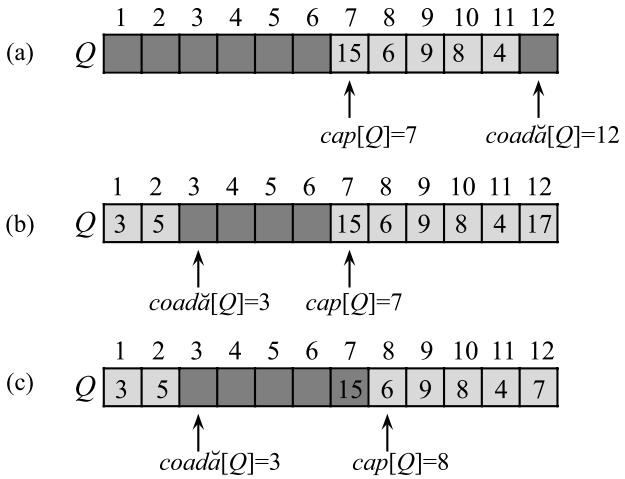


Figura 11.2 O coadă implementată utilizând un tablou $Q[1..12]$. Elementele cozii apar doar în poziţiile cu haşură deschisă. **(a)** Coada are 5 elemente, în locaţiile $Q[7..11]$. **(b)** Configuraţia cozii după apelurile PUNE-ÎN-COADĂ (Q,17), PUNE-ÎN-COADĂ(Q,3), PUNE-ÎN-COADĂ(Q,5). **(c)** Configuraţia cozii după ce apelul SCOATE-DIN-COADĂ(Q) returnează valoarea cheii 15, ce s-a aflat anterior în capul cozii. Noul cap are cheia 6.

PUNE-ÎN-COADĂ(Q, x)
1: $Q[coadă[Q]] \leftarrow x$
2: **dacă** $coadă[Q] = lung[Q]$ **atunci**
3: $\quad coadă[Q] \leftarrow 1$
4: **altfel**
5: $\quad coadă[Q] \leftarrow coadă[Q] + 1$

SCOATE-DIN-COADĂ(Q)
1: $x \leftarrow Q[cap[Q]]$
2: **dacă** $cap[Q] = lung[Q]$ **atunci**
3: $\quad cap[Q] \leftarrow 1$
4: **altfel**
5: $\quad cap[Q] \leftarrow cap[Q] + 1$
6: **returnează** x

În figura 11.2 sunt ilustrate efectele operaţiilor PUNE-ÎN-COADĂ şi SCOATE-DIN-COADĂ. Fiecare operaţie necesită un timp $O(1)$.

Exerciţii

11.1-1 Folosind ca model figura 11.1, ilustraţi efectul fiecăreia dintre operaţiile PUNE-ÎN-STIVĂ$(S, 4)$, PUNE-ÎN-STIVĂ$(S, 1)$, PUNE-ÎN-STIVĂ$(S, 3)$, SCOATE-DIN-STIVĂ(S), PUNE-ÎN-STIVĂ$(S, 8)$, SCOATE-DIN-STIVĂ(S) pe o stivă iniţial vidă, memorată într-un tablou $S[1..6]$.

11.1-2 Explicaţi cum se pot implementa două stive într-un singur tablou $A[1..n]$ astfel încât nici una dintre stive să nu aibă depăşire superioară, cu excepţia cazului în care numărul total de elemente din ambele stive (împreună) este n. Operaţiile PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ trebuie să funcţioneze într-un timp $O(1)$.

11.1-3 Folosind ca model figura 11.2, ilustraţi efectul fiecăreia dintre operaţiile PUNE-ÎN-COADĂ$(Q, 4)$, PUNE-ÎN-COADĂ$(Q, 1)$, PUNE-ÎN-COADĂ$(Q, 3)$, SCOATE-DIN-COADĂ(Q), PUNE-ÎN-COADĂ$(Q, 8)$ şi SCOATE-DIN-COADĂ (Q) pe o coadă Q, iniţial vidă, a memorată într-un tablou $Q[1..6]$.

11.1-4 Rescrieţi PUNE-ÎN-COADĂ şi SCOATE-DIN-COADĂ pentru a detecta depăşirile unei cozi.

11.1-5 În timp ce stiva permite inserarea şi ştergerea de elemente doar la un singur capăt, iar coada permite inserarea la un capăt şi ştergerea la celălalt capăt, o ***coadă completă*** permite inserări şi ştergeri la ambele capete. Scrieţi patru proceduri cu timpul de execuţie $O(1)$ pentru inserare de elemente şi ştergere de elemente la ambele capete ale unei cozi complete implementată printr-un tablou.

11.1-6 Arătaţi cum se poate implementa o coadă prin două stive. Analizaţi timpul de execuţie pentru operaţiile cozii.

11.1-7 Arătaţi cum se poate implementa o stivă prin două cozi. Analizaţi timpul de execuţie pentru operaţiile stivei.

11.2. Liste înlănţuite

O ***listă înlănţuită*** este o structură de date în care obiectele sunt aranjate într-o anumită ordine. Spre deosebire de tablou, în care ordinea este determinată de indicii tabloului, ordinea într-o listă înlănţuită este determinată de un pointer conţinut în fiecare obiect. Listele înlănţuite asigură o reprezentare mai simplă şi mai flexibilă pentru mulţimi dinamice, suportând (deşi nu neapărat eficient) toate operaţiile descrise la pagina 168.

După cum se arată în figura 11.3, fiecare element al unei ***liste dublu înlănţuite*** L este un obiect cu un câmp *cheie* şi alte două câmpuri pointer: *urm* (următor) şi *prec* (precedent). Obiectul poate conţine şi alte date. Fiind dat un element x din listă, $urm[x]$ indică spre succesorul său din lista înlănţuită, iar $prec[x]$ indică spre predecesorul său. Dacă $prec[x] =$ NIL atunci elementul x nu are nici un predecesor şi deci e primul element din listă, sau ***capul*** listei. Dacă $urm[x] =$NIL atunci elementul x nu are succesor şi deci este ultimul element sau ***coada*** listei. Un atribut $cap[L]$ referă primul element al listei. Dacă $cap[L] =$NIL, lista este vidă.

O listă poate avea una din următoarele forme. Ea poate fi simplu înlănţuită sau dublu înlănţuită, poate fi sortată sau nu, şi poate fi, sau nu, circulară. Dacă o listă este ***simplu înlănţuită*** vom omite pointerul *prec* din fiecare element. Dacă o listă este ***sortată***, ordinea din listă corespunde ordinii cheilor memorate în elementele listei; elementul minim este capul listei, iar elementul maxim este coada listei. Dacă lista este ***nesortată***, elementele pot apărea în orice ordine. Într-o ***listă circulară*** referinţa *prec* a capului listei referă coada iar referinţa *urm* a cozii listei referă capul ei. Astfel, lista poate fi privită ca un inel de elemente. În restul acestei secţiuni vom presupune că listele cu care lucrăm sunt nesortate şi dublu înlănţuite.

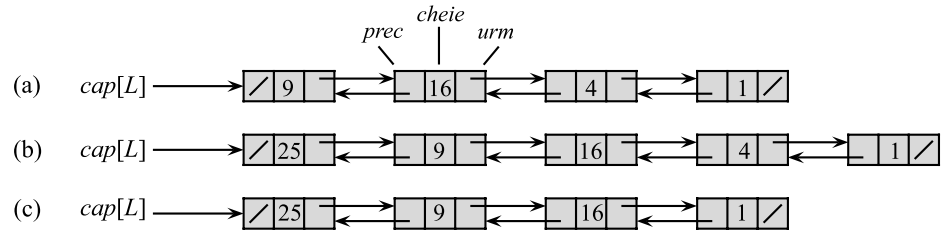


Figura 11.3 **(a)** O listă dublu înlănţuită L reprezentând mulţimea dinamică $\{1, 4, 9, 16\}$. Fiecare element din listă este un obiect având câmpuri pentru cheie şi doi pointeri (ilustraţi prin săgeţi) la elementul următor, respectiv anterior. Câmpul *urm* al cozii şi câmpul *prec* al capului sunt NIL, lucru indicat printr-un slash. Atributul $cap[L]$ referă capul listei. **(b)** După execuţia lui LISTĂ-INSEREAZĂ(L, x), unde $cheie[x] = 25$, capul listei înlănţuite este un nou obiect având cheia 25. Acest nou obiect referă vechiul cap având cheia 9. **(c)** Rezultatul apelului LISTĂ-ŞTERGE(L, x), unde x referă obiectul având cheia 4.

Căutarea într-o listă înlănţuită

Procedura LISTĂ-CAUTĂ(L, k) găseşte primul element având cheia k din lista L printr-o căutare liniară simplă, returnând pointerul la acest element. Dacă în listă nu apare nici un obiect având cheia k, atunci se returnează NIL. Pentru lista înlănţuită din figura 11.3(a), apelul LISTĂ-CAUTĂ$(L, 4)$ returnează un pointer la al treilea element, iar apelul LISTĂ-CAUTĂ$(L, 7)$ returnează NIL.

LISTĂ-CAUTĂ(L, x)
1: $x \leftarrow cap[L]$
2: **cât timp** $x \neq$ NIL şi $cheie[x] \neq k$ **execută**
3: $\quad x \leftarrow urm[x]$
4: **returnează** x

Pentru a căuta într-o listă având n obiecte, procedura LISTĂ-CAUTĂ necesită un timp $\Theta(n)$ în cazul cel mai defavorabil, deoarece va trebui să caute în toată lista.

Inserarea într-o listă înlănţuită

Fiind dat un element x al cărui câmp *cheie* a fost iniţializat, procedura LISTĂ-INSEREAZĂ îl plasează pe x în faţa listei înlănţuite, după cum se poate observa în figura 11.3(b).

LISTĂ-INSEREAZĂ(L, x)
1: $urm[x] \leftarrow cap[L]$
2: **dacă** $cap[L] \neq$ NIL **atunci**
3: $\quad prec[cap[L]] \leftarrow x$
4: $cap[L] \leftarrow x$
5: $prec[x] \leftarrow$ NIL

Timpul de execuţie pentru LISTĂ-INSEREAZĂ pe o listă cu n elemente este $O(1)$.

Ştergerea dintr-o listă înlănţuită

Procedura LISTĂ-ŞTERGE elimină un element x dintr-o listă înlănţuită L. Pentru aceasta, trebuie transmis ca argument un pointer spre x şi x va fi scos din listă, actualizându-se pointerii. Dacă dorim să ştergem un element cu o cheie dată, mai întâi trebuie să apelăm LISTĂ-CAUTĂ pentru a afla pointerul spre acel element.

LISTĂ-ŞTERGE(L, x)
1: **dacă** $prec[x] \neq$ NIL **atunci**
2: $\quad urm[prec[x]] \leftarrow urm[x]$
3: **altfel**
4: $\quad cap[L] \leftarrow urm[x]$
5: **dacă** $urm[x] \neq$ NIL **atunci**
6: $\quad prec[urm[x]] \leftarrow prec[x]$

Figura 11.3(c) ilustrează cum se şterge un element dintr-o listă înlănţuită. LISTĂ-ŞTERGE se execută într-un timp $O(1)$, dar dacă dorim să ştergem un element având o cheie dată, timpul necesar pentru cazul cel mai defavorabil este $\Theta(n)$, pentru că înainte trebuie să apelăm LISTĂ-CAUTĂ.

Santinele

Codul pentru LISTĂ-ŞTERGE ar fi mai simplu dacă am ignora condiţiile extreme de la capul şi coada listei.

LISTĂ-ŞTERGE$'(L, x)$
1: $urm[prec[x]] \leftarrow urm[x]$
2: $prec[urm[x]] \leftarrow prec[x]$

O ***santinelă*** este un obiect fictiv care ne permite să simplificăm condiţiile de la extreme. De exemplu, să presupunem că în lista L avem la dispoziţie un obiect $nil[L]$, care reprezintă NIL, dar care are toate câmpurile unui element al listei. De câte ori avem o referinţă la NIL în codul listei, o vom înlocui cu o referinţă la santinela $nil[L]$. După cum se observă în figura 11.4, santinela transformă o listă dublu înlănţuită într-o listă circulară, cu santinela $nil[L]$ plasată între cap şi coadă; câmpul $urm[nil[L]]$ indică spre capul listei, iar $prec[nil[L]]$ indică spre coadă. Similar, atât câmpul urm al cozii cât şi câmpul $prec$ al capului indică spre $nil[L]$. Deoarece $urm[nil[L]]$ indică spre cap, putem elimina atributul $cap[L]$, înlocuind referinţele către el cu referinţe către $urm[nil[L]]$. O listă vidă va conţine doar santinela, deoarece atât $urm[nil[L]]$ cât şi $prec[nil[L]]$ pot fi setate pe $nil[L]$.

Codul pentru LISTĂ-CAUTĂ rămâne la fel ca înainte, dar cu referinţele spre NIL şi $cap[L]$ schimbate după cum s-a precizat anterior.

LISTĂ-CAUTĂ$'(L, k)$
1: $x \leftarrow urm[nil[L]]$
2: **cât timp** $x \neq nil[L]$ şi $cheie[x] \neq k$ **execută**
3: $\quad x \leftarrow urm[x]$
4: **returnează** x

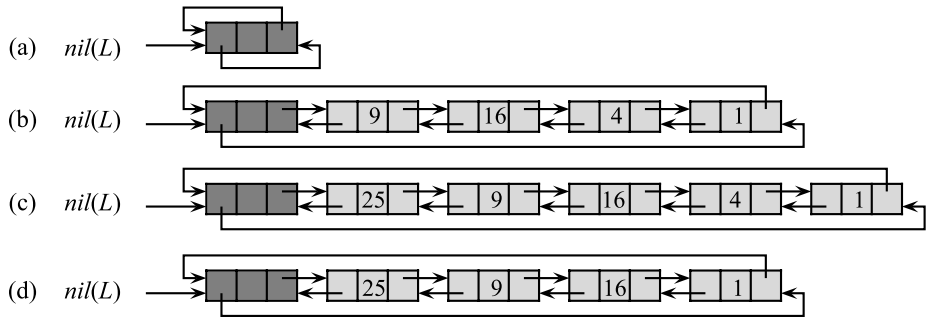


Figura 11.4 O listă înlănţuită L ce foloseşte o santinelă $nil[L]$ (haşurată cu negru) este o listă dublu înlănţuită obişnuită, transformată într-o listă circulară cu $nil[L]$ pus între cap şi coadă. Atributul $cap[L]$ nu mai este necesar, deoarece capul listei poate fi accesat prin $urm[nil[L]]$. **(a)** O listă vidă. **(b)** Lista înlănţuită din figura 11.3(a), având cheia 9 în capul listei şi cheia 1 în coada listei. **(c)** Lista după execuţia procedurii LISTĂ-INSEREAZĂ′(L, x), unde $cheie[x] = 25$. Noul obiect devine capul listei. **(d)** Lista după ştergerea obiectului având cheia 1. Noua coadă este obiectul având cheia 4.

Folosim procedura (cu două linii de cod) LISTĂ-ŞTERGE′ pentru a şterge un element din listă. Vom folosi următoarea procedură pentru inserarea unui element în listă.

LISTĂ-INSEREAZĂ′(L, x)
1: $urm[x] \leftarrow urm[nil[L]]$
2: $prec[urm[nil[L]]] \leftarrow x$
3: $urm[nil[L]] \leftarrow x$
4: $prec[x] \leftarrow nil[L]$

Figura 11.4 ilustrează efectele procedurilor LISTĂ-INSEREAZĂ′ şi LISTĂ-ŞTERGE′ pe un exemplu.

Rareori santinelele reduc marginile asimptotice de timp pentru operaţiile structurilor de date, dar pot reduce factorii constanţi. Câştigul pe care îl reprezintă folosirea santinelelor în cadrul ciclurilor este de obicei mai mult legat de claritatea codului, decât de viteză; de exemplu, codul pentru lista înlănţuită este simplificat prin utilizarea santinelelor, dar câştigăm doar un timp $O(1)$ în procedurile LISTĂ-INSEREAZĂ′ şi LISTĂ-ŞTERGE′. Oricum, în alte situaţii, folosirea santinelelor ajută la restrângerea codului dintr-un ciclu, reducându-se astfel coeficientul, de exemplu, pentru n sau n^2 din timpul de execuţie.

Santinelele trebuie folosite cu discernământ. Dacă avem multe liste mici, spaţiul suplimentar folosit de santinelele acestor liste poate reprezenta o risipă semnificativă de memorie. Astfel, în cartea de faţă santinelele se vor folosi doar acolo unde se obţine o simplificare semnificativă a codului.

Exerciţii

11.2-1 Se poate implementa operaţia INSEREAZĂ pentru mulţimi dinamice pe o listă simplu înlănţuită cu un timp de execuţie $O(1)$? Dar operaţia ŞTERGE?

11.2-2 Implementaţi o stivă folosind o listă simplu înlănţuită L. Operaţiile PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ ar trebui să necesite tot un timp $O(1)$.

11.2-3 Implementaţi o coadă folosind o listă simplu înlănţuită L. Operaţiile PUNE-ÎN-COADĂ şi SCOATE-DIN-COADĂ ar trebui să necesite tot un timp $O(1)$.

11.2-4 Implementaţi operaţiile specifice dicţionarelor INSEREAZĂ, ŞTERGE şi CAUTĂ folosind liste simplu înlănţuite, respectiv circulare. Care sunt timpii de execuţie pentru aceste proceduri?

11.2-5 Operaţia pentru mulţimi dinamice REUNEŞTE are ca intrare două mulţimi disjuncte S_1 şi S_2 şi returnează mulţimea $S = S_1 \cup S_2$ ce conţine toate elementele din S_1 şi S_2. Mulţimile S_1 şi S_2 sunt de obicei distruse de către operaţie. Arătaţi cum se poate realiza REUNEŞTE într-un timp $O(1)$ folosind o structură de date de tip listă, adecvată.

11.2-6 Scrieţi o procedură care interclasează două liste simplu înlănţuite sortate într-o singură listă simplu înlănţuită sortată fără a folosi santinele. Apoi, scrieţi o procedură similară folosind o santinelă având cheia ∞ pentru a marca sfârşitul fiecărei liste. Comparaţi simplitatea codului pentru fiecare din cele două proceduri.

11.2-7 Creaţi o procedură nerecursivă de timp de execuţie $\Theta(n)$ care inversează o listă simplu înlănţuită având n elemente. Procedura va folosi un spaţiu suplimentar de memorie de mărime constantă.

11.2-8 $\star$ Explicaţi cum se pot implementa liste dublu înlănţuite folosind o singură valoare referinţă $np[x]$ pentru fiecare obiect, în locul celor două valori uzuale (*urm* şi *prec*). Se presupune că toate valorile pointerilor pot fi interpretate ca şi întregi pe k biţi şi se defineşte $np[x]$ ca fiind $np[x] = urm[x]$ XOR $prec[x]$, "sau exclusiv" pe k biţi dintre $urm[x]$ şi $prec[x]$. (Valoarea NIL e reprezentată de 0.) Acordaţi atenţie descrierii informaţiei necesare pentru accesarea capului listei. Arătaţi cum pot fi implementate operaţiile CAUTĂ, INSEREAZĂ şi ŞTERGE pe o astfel de listă. De asemenea, arătaţi cum se poate inversa o astfel de listă într-un timp $O(1)$.

11.3. Implementarea pointerilor şi obiectelor

Cum implementăm pointerii şi obiectele în limbaje cum ar fi Fortran, care nu asigură lucrul cu astfel de date? În această secţiune, vom arăta două modalităţi de implementare a structurilor de date înlănţuite fără un tip de date pointer explicit. Vom sintetiza obiectele şi pointerii din tablouri şi indici de tablouri.

O reprezentare multi-tablou a obiectelor

O colecţie de obiecte care au aceleaşi câmpuri se poate reprezenta utilizând câte un tablou pentru fiecare câmp. De exemplu, figura 11.5 ilustrează cum se poate implementa lista înlănţuită din figura 11.3(a) prin trei tablouri. Tabloul *cheie* reţine valorile curente ale cheilor din mulţimea dinamică, iar pointerii sunt memoraţi în tablourile *urm* şi *prec*. Pentru un indice de tablou x dat, $cheie[x], urm[x]$ şi $prec[x]$ reprezintă un obiect în lista înlănţuită. Într-o astfel de implementare, un pointer x este un simplu indice în tablourile *cheie, urm* şi *prec*.

În figura 11.3(a), obiectul având cheia 4 urmează după obiectul având cheia 16 în lista înlănţuită. În figura 11.5, cheia 4 apare în $cheie[2]$, iar cheia 16 apare în $cheie[5]$, deci vom avea $urm[5] = 2$ şi $prec[2] = 5$. Deşi constanta NIL apare în câmpul *urm* al cozii şi în câmpul

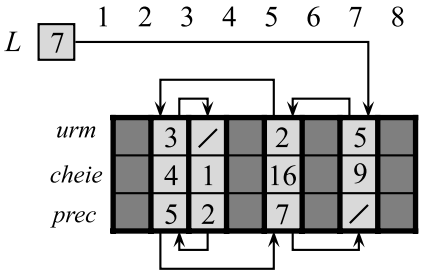


Figura 11.5 Lista înlănţuită din figura 11.3(a) reprezentată prin tablourile *cheie, urm* şi *prec*. Fiecare porţiune verticală a tablourilor reprezintă un singur obiect. Pointerii memoraţi corespund indicilor tablourilor ilustraţi în partea de sus; săgeţile arată cum trebuie interpretaţi aceşti indici. Poziţiile cu haşură deschisă ale obiectelor conţin elementele listei. Variabila L conţine indicele capului.

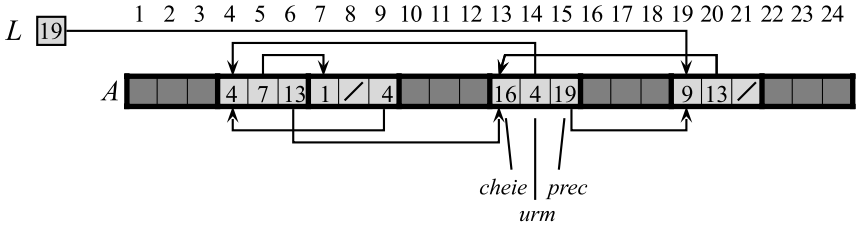


Figura 11.6 Lista înlănţuită din figura 11.3(a) şi figura 11.5 reprezentată printr-un singur tablou A. Fiecare element al listei este un obiect care ocupă un subtablou continuu de lungime 3 în cadrul tabloului. Cele trei câmpuri *cheie, urm* şi *prec* corespund deplasamentelor 0, 1 şi respectiv 2. Pointerul la un obiect este indicele primului element al obiectului. Obiectele conţinând elementele listei sunt haşurate deschis, iar săgeţile indică ordinea în listă.

prec al capului, de obicei vom folosi un întreg (ca 0 sau -1) care nu poate reprezenta un indice valid în tablouri. O variabilă *L* memorează indicele capului listei.

În pseudocodul nostru, am folosit paranteze pătrate pentru a nota atât indexarea într-un tablou cât şi selecţia unui câmp (atribut) aparţinând unui obiect. În ambele cazuri, semnificaţia lui $cheie[x], urm[x]$ şi $prec[x]$ este consistentă relativ la implementare.

O reprezentare a obiectelor printr-un tablou unic

Cuvintele din memoria unui calculator sunt în mod obişnuit adresate prin întregi între 0 şi $M-1$, unde M este un număr întreg suficient de mare. În multe limbaje de programare, un obiect ocupă o mulţime contiguă de locaţii în memoria calculatorului. Un pointer este pur şi simplu adresa primei locaţii de memorie a obiectului iar celelalte locaţii de memorie din cadrul obiectului pot fi indexate adăugând un deplasament la acest pointer.

În medii de programare care nu au tipuri de date pointer explicite putem folosi aceeaşi strategie pentru implementarea obiectelor. De exemplu, figura 11.6 ilustrează cum se poate folosi un singur tablou *A* pentru a memora lista înlănţuită din figura 11.3(a) şi figura 11.5. Un obiect ocupă un subtablou contiguu $A[j..k]$. Fiecare câmp al obiectului corespunde unui deplasament din intervalul $0..k-j$, iar un pointer la obiect este indicele *j*. În figura 11.6, deplasamentele

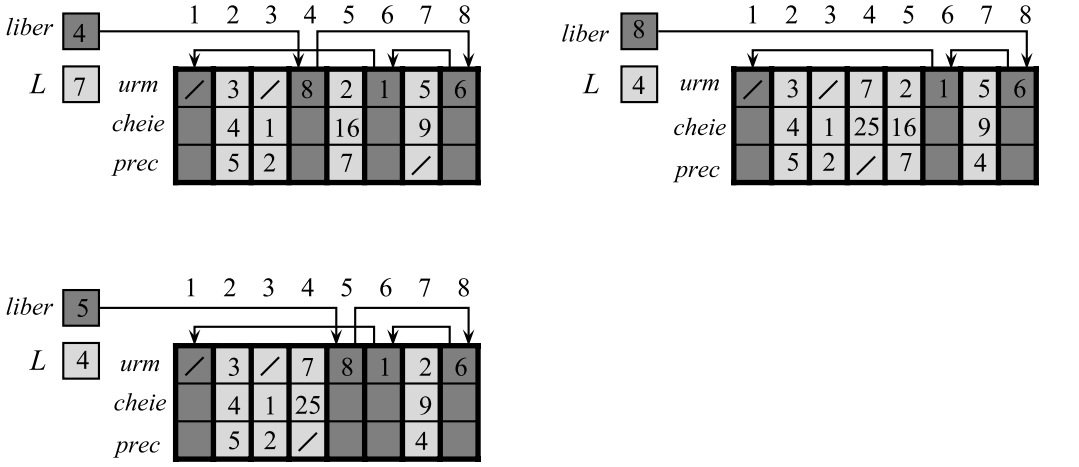


Figura 11.7 Efectul procedurilor ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT. **(a)** Lista din figura 11.5 (partea cu haşură deschisă) şi o listă liberă (partea cu haşură închisă). Săgeţile indică structura listei libere. **(b)** Rezultatul apelului ALOCĂ-OBIECT() (care returnează cheia 4), setând $cheie[4]$ la 25 şi al apelului LISTĂ-INSEREAZĂ$(L, 4)$. Noul cap al listei libere este obiectul 8, care a fost $urm[4]$ în lista liberă. **(c)** După execuţia lui LISTĂ-ŞTERGE$(L, 5)$, apelăm ELIBEREAZĂ-OBIECT(5). Obiectul 5 devine noul cap al listei libere, cu obiectul 8 urmându-i în listă.

corespunzătoare lui *cheie*, *urm* şi *prec* sunt 0, 1, respectiv 2. Pentru a citi valoarea lui $prec[i]$ pentru un pointer i dat, adăugăm la valoarea i a pointerului deplasamentul 2, citind astfel $A[i+2]$.

Reprezentarea printr-un singur tablou este flexibilă în sensul că permite memorarea de obiecte cu lungimi diferite în acelaşi tablou. Problema gestionării unei astfel de colecţii eterogene de obiecte este mult mai dificilă decât problema gestionării unei colecţii omogene, în care toate obiectele au aceleaşi câmpuri. Deoarece majoritatea structurilor de date pe care le luăm în considerare sunt compuse din elemente omogene, pentru scopurile noastre va fi suficient să folosim reprezentarea multi-tablou a obiectelor.

Alocarea şi eliberarea obiectelor

Pentru a insera o cheie într-o mulţime dinamică reprezentată printr-o listă dublu înlănţuită, trebuie să alocăm un pointer la un obiect neutilizat la momentul respectiv în reprezentarea listei înlănţuite. Prin urmare, este util să ţinem evidenţa obiectelor care nu sunt folosite la un moment dat, în reprezentarea listei înlănţuite; astfel, ori de câte ori avem nevoie de un nou obiect, acesta va fi preluat din lista de obiecte nefolosite. În unele sisteme, un aşa numit "***garbage collector***" (colector de reziduuri) este responsabil cu determinarea obiectelor neutilizate. Oricum, sunt multe aplicaţii, suficient de simple astfel încât îşi pot asuma responsabilitatea de a returna un obiect neutilizat către gestionarul de obiecte nefolosite. Vom studia acum problema alocării şi eliberării (sau dealocării) obiectelor omogene folosind, de exemplu, o listă dublu înlănţuită reprezentată prin tablouri multiple.

Să presupunem că tablourile din reprezentarea multi-tablou au lungimea m şi că la un moment dat mulţimea dinamică conţine $n \leq m$ elemente. Atunci n obiecte vor reprezenta elementele

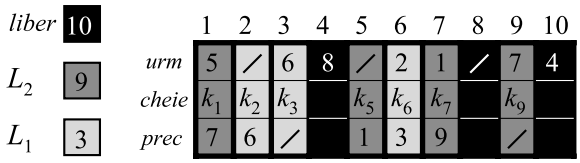


Figura 11.8 Două liste înlănţuite L_1 (porţiunea cu haşură deschisă) şi L_2 (cu haşură medie) şi o listă liberă interconectată (cu haşură închisă).

curente din mulţimea dinamică şi restul de $m - n$ obiecte sunt ***libere***; obiectele libere pot fi folosite pentru reprezentarea elementelor ce se vor insera în viitor în mulţimea dinamică.

Vom păstra obiectele libere într-o listă simplu înlănţuită, pe care o vom numi ***listă liberă***. Lista liberă foloseşte doar tabloul *urm*, care memorează pointerii *urm* din listă. Capul listei libere este memorat într-o variabilă globală *liber*. În cazul în care mulţimea dinamică reprezentată de lista înlănţuită L este nevidă, lista liberă se poate întreţese cu lista L, după cum se observă în figura 11.7. Observaţi că fiecare obiect din reprezentare este fie în lista L, fie în lista liberă, dar nu în amândouă.

Lista liberă este o stivă: următorul obiect alocat este cel mai recent eliberat. Putem folosi o implementare a operaţiilor stivei PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ folosind o listă pentru a construi procedurile pentru alocarea şi, respectiv, eliberarea obiectelor. Presupunem că variabila globală *liber* folosită în următoarele proceduri referă primul element al listei libere.

ALOCĂ-OBIECT()
1: **dacă** *liber* = NIL **atunci**
2: **eroare** "depăşire de spaţiu"
3: **altfel**
4: $x \leftarrow liber$
5: $liber \leftarrow urm[x]$
6: **returnează** x

ELIBEREAZĂ-OBIECT(x)
1: $urm[x] \leftarrow liber$
2: $liber \leftarrow x$

Iniţial lista liberă conţine toate cele n obiecte nealocate. În cazul în care lista liberă este epuizată, procedura ALOCĂ-OBIECT semnalează o eroare. În mod curent se poate folosi o singură listă liberă care să servească mai multe liste înlănţuite. Figura 11.8 ilustrează două liste înlănţuite şi o listă liberă, interconectate prin tablourile *cheie*, *urm* şi *prec*.

Cele două proceduri se execută într-un timp $O(1)$, ceea ce le face eficiente. Ele pot fi modificate astfel încât să funcţioneze pentru orice colecţie omogenă de obiecte, lăsând unul din câmpurile obiectului să se comporte ca un câmp *urm* în lista liberă.

Exerciţii

11.3-1 Desenaţi o imagine a şirului $\langle 13, 4, 8, 19, 5, 11\rangle$ memorată ca o listă dublu înlănţuită folosind reprezentarea multi-tablou. Realizaţi acelaşi lucru pentru reprezentarea cu un tablou unic.

11.3-2 Scrieţi procedurile ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT pentru o colecţie omogenă de obiecte implementată printr-o reprezentare cu un tablou unic.

11.3-3 De ce nu este necesar să setăm sau să resetăm câmpurile *prec* ale obiectelor din implementarea procedurilor ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT?

11.3-4 De multe ori este de dorit să păstrăm toate elementele unei liste dublu înlănţuite într-o zonă compactă la memorare, folosind, de exemplu, primele m locaţii din reprezentarea multi-tablou. (Aceasta este situaţia într-un mediu de calcul cu memorie virtuală, paginată.) Explicaţi, cum se pot implementa procedurile ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT astfel încât reprezentarea să fie compactă. Presupuneţi că nu există pointeri la elementele listei înlănţuite în afara listei înseşi. (*Indicaţie*: Folosiţi implementarea stivei printr-un tablou.)

11.3-5 Fie L o listă dublu înlănţuită de lungime m memorată în tablourile *cheie*, *prec* şi *urm* de lungime n. Să presupunem că aceste tablouri sunt gestionate de procedurile ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT, care păstrează o listă liberă dublu înlănţuită F. Mai presupunem că din cele n locaţii, sunt exact m în lista L şi $n-m$ în lista liberă. Scrieţi o procedură COMPACTEAZĂ-LISTĂ(L, F) care, fiind date lista L şi lista liberă F, mută elementele din L astfel încât ele să ocupe poziţiile $1, 2, ..., m$ din tablou şi ajustează lista liberă F astfel încât să rămână corectă, ocupând poziţiile $m+1, m+2, ..., n$ din tablou. Timpul de execuţie pentru procedura scrisă trebuie să fie $\Theta(m)$ şi ea trebuie să utilizeze doar un spaţiu suplimentar constant. Daţi o argumentare atentă a corectitudinii procedurii scrise.

11.4. Reprezentarea arborilor cu rădăcină

Metodele pentru reprezentarea listelor date în secţiunea precedentă se extind la orice structură omogenă de date. În această secţiune, vom discuta în mod special problema reprezentării arborilor cu rădăcină prin structuri de date înlănţuite. Vom studia mai întâi arborii binari şi apoi vom prezenta o metodă pentru arbori cu rădăcină în care nodurile pot avea un număr arbitrar de descendenţi.

Vom reprezenta fiecare nod al arborilor printr-un obiect. Ca în cazul listelor înlănţuite, presupunem că fiecare nod conţine un câmp *cheie*. Restul câmpurilor care prezintă interes sunt pointeri către celelalte noduri şi pot varia în funcţie de tipul arborelui.

Arbori binari

După cum se observă în figura 11.9 câmpurile *p*, *stânga* şi *dreapta* se folosesc pentru a memora pointerii pentru părinte, descendentul stâng şi descendentul drept al fiecărui nod din arborele binar T. Dacă $p[x] = \text{NIL}$, atunci x este rădăcina. Dacă nodul x nu are descendent stâng, atunci *stânga*$[x] = \text{NIL}$ şi similar pentru descendentul drept. Rădăcina întregului arbore T este referită de atributul *rădăcină*$[T]$. Dacă *rădăcină*$[T] = \text{NIL}$, atunci arborele este vid.

Arbori cu rădăcină cu număr nelimitat de ramuri

Schema pentru reprezentarea arborilor binari poate fi extinsă la orice clasă de arbori în care numărul de descendenţi ai fiecărui nod nu depăşeşte o constantă k: vom înlocui câmpurile *stânga*

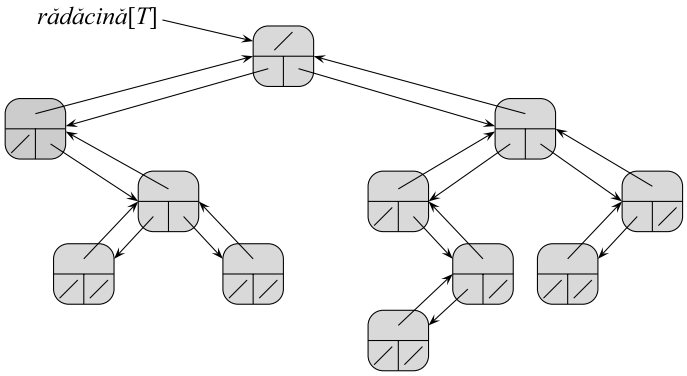


Figura 11.9 Reprezentarea unui arbore binar T. Fiecare nod x are câmpurile $p[x]$ (partea de sus), *stânga*$[x]$ (partea din stânga jos) şi *dreapta*$[x]$ (partea din dreapta jos). Câmpurile *cheie* nu sunt ilustrate.

şi *dreapta* cu $fiu_1, fiu_2, ..., fiu_k$. Această schemă nu mai funcţionează atunci când numărul de descendenţi ai unui nod este nemărginit, deoarece nu ştim câte câmpuri (tablouri în reprezentarea multi-tablou) să alocăm în avans. Mai mult, chiar dacă numărul k de descendenţi este mărginit de o constantă mare, dar majoritatea nodurilor au un număr mic de descendenţi, vom irosi multă memorie.

Din fericire, există o schemă "inteligentă" de a folosi arbori binari pentru a reprezenta arbori cu număr arbitrar de descendenţi. Ea are avantajul că utilizează un spaţiu de $O(n)$ pentru orice arbore cu rădăcină şi cu n noduri. ***Reprezentarea descendent-stâng, frate-drept*** este ilustrată în figura 11.10. Fiecare nod conţine un pointer spre părinte p, iar *rădăcină*$[T]$ referă rădăcina arborelui T. În loc să avem un pointer spre fiecare descendent, fiecare nod x are doar doi pointeri:

1. *fiu-stâng*$[x]$ referă cel mai din stânga descendent al nodului x şi
2. *frate-drept*$[x]$ referă fratele lui x, cel mai apropiat spre dreapta.

Dacă nodul x nu are descendenţi atunci *fiu-stâng*$[x] =$ NIL, iar dacă nodul x este cel mai din dreapta descendent al părintelui său atunci *frate-drept*$[x] =$ NIL.

Alte reprezentări ale arborilor

Uneori reprezentăm arborii cu rădăcină şi în alte moduri. În capitolul 7, de exemplu, am reprezentat un ansamblu, care se bazează pe un arbore binar complet, printr-un singur tablou plus un indice. Arborii care apar în capitolul 22 sunt traversaţi numai spre rădăcină, de aceea doar pointerii spre părinţi sunt prezenţi; nu există pointeri spre descendenţi. Sunt posibile şi multe alte scheme. Care este cea mai bună schemă care depinde de aplicaţie.

Exerciţii

11.4-1 Desenaţi arborele binar având rădăcina la indicele 6, care este reprezentat de următoarele câmpuri.

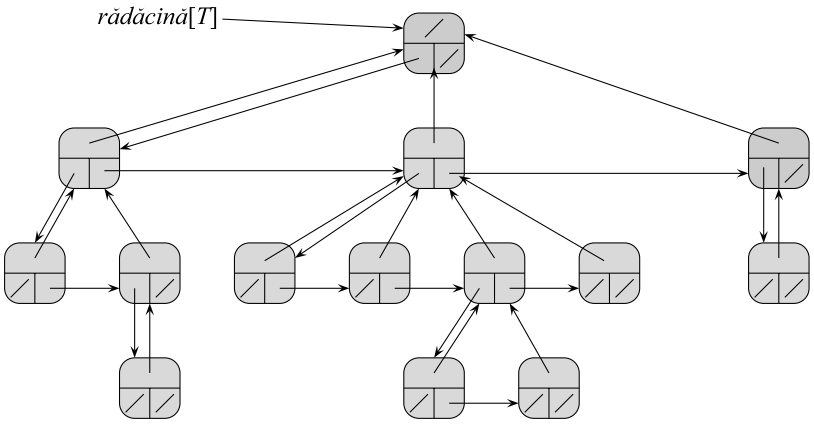


Figura 11.10 Reprezentarea descendent-stâng, frate-drept a unui arbore T. Fiecare nod x are câmpurile $p[x]$ (partea de sus), *fiu-stâng*$[x]$ (partea din stânga jos) şi *frate-drept*$[x]$ (partea din dreapta jos). Cheile nu sunt ilustrate.

indice	*cheie*	*stânga*	*dreapta*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

11.4-2 Scrieţi o procedură recursivă cu timp de execuţie $O(n)$ care, fiind dat un arbore binar cu n noduri, tipăreşte cheia fiecărui nod din arbore.

11.4-3 Scrieţi o procedură nerecursivă cu timp de execuţie $O(n)$ care, fiind dat un arbore binar cu n noduri, tipăreşte cheia fiecărui nod din arbore. Folosiţi o stivă ca o structură auxiliară de date.

11.4-4 Scrieţi o procedură cu timp de execuţie $O(n)$ care tipăreşte toate cheile unui arbore arbitrar cu rădăcină având n noduri, dacă arborele este memorat folosind reprezentarea descendent-stâng, frate-drept.

11.4-5 $\star$ Scrieţi o procedură nerecursivă cu timp de execuţie $O(n)$ care, fiind dat un arbore binar cu n noduri, tipăreşte cheile fiecărui nod. Folosiţi numai un spaţiu suplimentar constant în afara arborelui însuşi şi nu modificaţi arborele, nici măcar temporar, în timpul procedurii.

11.4-6 $\star$ Reprezentarea descendent-stâng, frate-drept a unui arbore arbitrar cu rădăcină foloseşte trei pointeri în fiecare nod: *fiu-stâng, frate-drept* şi *părinte*. Din fiecare nod se pot atinge şi identifica părintele şi toţi descendenţii nodului. Arătaţi cum se poate obţine acelaşi efect, folosind doar doi pointeri şi o valoare booleană în fiecare nod.

Probleme

11-1 Comparaţii între liste

Pentru fiecare din cele patru tipuri de liste din tabelul următor, care este timpul asimptotic de execuţie în cazul cel mai defavorabil pentru fiecare dintre operaţiile pe mulţimi dinamice specificate în tabelul următor?

	nesortată, simplu înlănţuită	sortată, simplu înlănţuită	nesortată, dublu înlănţuită	sortată, dublu înlănţuită
CAUTĂ(L, k)				
INSEREAZĂ(L, x)				
ŞTERGE(L, x)				
SUCCESOR(L, x)				
PREDECESOR(L, x)				
MINIM(L)				
MAXIM(L)				

11-2 Absambluri interclasate folosind liste înlănţuite

Un ***ansamblu interclasat*** are următoarele operaţii: CREEAZĂ-ANSAMBLU (care creează un ansamblu interclasat vid), INSEREAZĂ, MINIM, EXTRAGE-MIN şi REUNEŞTE. Arătaţi cum se pot implementa ansamblurile interclasate folosind liste înlănţuite în fiecare din următoarele cazuri. Încercaţi să realizaţi fiecare operaţie cât mai eficient posibil. Analizaţi timpul de execuţie în termenii dimensiunii mulţimii (mulţimilor) dinamice cu care se operează. ***a.*** Listele sunt sortate.

b. Listele nu sunt sortate. ***c.*** Listele nu sunt sortate, iar mulţimile dinamice ce se interclasează sunt disjuncte.

11-3 Căutarea într-o listă sortată compactă

Exerciţiul 11.3-4 cerea modul în care putem menţine o listă având n elemente, compactă în primele n poziţii ale unui tablou. Vom presupune că toate cheile sunt distincte şi că lista compactă este de asemenea sortată, adică, $cheie[i] < cheie[urm[i]]$, oricare ar fi $i = 1, 2, ..., n$ pentru care $urm[i] \neq$ NIL. Cu aceste presupuneri ne aşteptăm ca următorul algoritm aleator să efectueze căutarea în listă într-un timp $O(n)$.

LISTĂ-COMPACTĂ-CAUTĂ(L, k)

```
1: i ← cap[L]
2: n ← lung[L]
3: cât timp i ≠ NIL şi cheie[i] ≤ k execută
4:    j ← RANDOM(1, n)
5:    dacă cheie[i] < cheie[j] şi cheie[j] < k atunci
6:       i ← j
7:    i ← urm[i]
8:    dacă cheie[i] = k atunci
9:       returnează i
10: returnează NIL
```

Dacă ignorăm liniile 4–6 din procedură, atunci avem algoritmul uzual pentru căutarea într-o listă înlănţuită sortată, în care indicele i referă pe rând fiecare poziţie din listă. Liniile 4–6 încearcă să sară în avans la o poziţie j aleasă aleator. Un astfel de salt este benefic dacă $cheie[j]$ este mai mare decât $cheie[i]$ şi mai mică decât k; într-un astfel de caz j marchează o poziţie în listă prin care i va trebui să treacă în timpul unei căutări obişnuite în listă. Deoarece lista este compactă, ştim că orice alegere a lui j între 1 şi n referă un obiect din listă şi nu o poziţie în lista liberă.

a. De ce presupunem în LISTĂ-COMPACTĂ-CAUTĂ că toate cheile sunt distincte? Argumentaţi că salturile aleatoare nu ajută neapărat în cazul asimptotic, dacă lista conţine dubluri ale cheilor.

Performanţa procedurii LISTĂ-COMPACTĂ-CAUTĂ poate fi analizată împărţind execuţia sa în două faze. În timpul primei faze, ignorăm orice avans realizat în determinarea lui k, ce se realizează în liniile 7–9. Deci, faza 1 constă doar în avansul în listă prin salturi aleatoare. Asemănător, faza 2 ignoră avansul ce se realizează în liniile 4–6 şi astfel va opera ca o căutare liniară obişnuită.

Fie X_t variabila aleatoare care descrie distanţa în lista înlănţuită (şi anume, prin înlănţuirea pointerilor urm) de la poziţia i la cheia dorită k după t iteraţii ale fazei 1.

b. Argumentaţi că timpul de execuţie dorit pentru LISTĂ-COMPACTĂ-CAUTĂ este $O(t + E[X_t])$, pentru orice $t \geq 0$.

c. Arătaţi că $E[X_t] \leq \sum_{r=1}^{n}(1 - r/n)^t$. (*Indicaţie*: Folosiţi relaţia (6.28).)

d. Arătaţi că $\sum_{r=1}^{n-1} r^t \leq n^{t+1}/(t+1)$.

e. Demonstraţi că $E[X_t] \leq n/(t+1)$ şi explicaţi de ce această formulă este intuitivă.

f. Arătaţi că LISTĂ-COMPACTĂ-CAUTĂ se execută într-un timp de $O(\sqrt{n})$.

Note bibliografice

Aho, Hopcroft şi Ullman [5] şi Knuth [121] sunt referinţe excelente pentru structuri de date elementare. Gonnet [90] furnizează date experimentale asupra performanţelor operaţiilor pentru multe structuri de date.

Originea stivelor şi cozilor ca structuri de date în informatică este neclară, deoarece noţiunile corespunzătoare existau deja în matematică şi în prelucrarea tradiţională a documentelor înainte de introducerea calculatoarelor digitale. Knuth [121] îl citează pe A. M. Turing pentru folosirea stivelor la legarea subrutinelor în 1947.

De asemenea, structurile de date bazate pe pointeri par a fi o "invenţie din folclor". Conform lui Knuth, pointerii se pare că erau utilizaţi pe calculatoarele timpurii cu memorie pe tambur magnetic. Limbajul A-1, dezvoltat de G. M. Hopper în 1951 reprezenta formulele algebrice ca arbori binari. Knuth creditează limbajul IPL-II, dezvoltat în 1956 de A. Newell, J. C. Shaw şi H. A. Simon, cu recunoaşterea importanţei şi promovarea utilizării pointerilor. Limbajul lor IPL-III, dezvoltat în 1957, include operaţii explicite asupra stivelor.

12 Tabele de dispersie

Multe aplicaţii necesită o mulţime dinamică pentru care să se aplice numai operaţiile specifice pentru dicţionare INSEREAZĂ, CAUTĂ şi ŞTERGE. De exemplu, un compilator pentru un limbaj de programare întreţine o tabelă de simboluri, în care cheile elementelor sunt şiruri arbitrare de caractere ce corespund identificatorilor din limbaj. O tabelă de dispersie este o structură eficientă de date pentru implementarea dicţionarelor. Deşi căutarea unui element într-o tabelă de dispersie poate necesita la fel de mult timp ca şi căutarea unui element într-o listă înlănţuită – un timp $\Theta(n)$ în cazul cel mai defavorabil – în practică, dispersia funcţionează extrem de bine. Pe baza unor ipoteze rezonabile, timpul preconizat pentru căutarea unui element într-o tabelă de dispersie este $O(1)$.

O tabelă de dispersie este o generalizare a noţiunii mai simple de tablou. Adresarea directă într-un tablou foloseşte abilitatea noastră de a examina o poziţie arbitrară în tablou într-un timp $O(1)$. Secţiunea 12.1 discută în detaliu despre adresarea directă. Adresarea directă este aplicabilă în cazul în care ne putem permite să alocăm un tablou care are câte o poziţie pentru fiecare cheie posibilă.

Când numărul cheilor memorate efectiv este relativ mic faţă de numărul total de chei posibile, tabelele de dispersie devin o alternativă eficientă la adresarea directă într-un tablou, deoarece o tabelă de dispersie foloseşte în mod normal un tablou de mărime proporţională cu numărul de chei memorate efectiv. În loc să folosim direct cheia ca indice în tablou, indicele este *calculat* pe baza cheii. În secţiunea 12.2 sunt prezentate ideile principale, iar în secţiunea 12.3 se descrie cum pot fi calculaţi indicii din tablou pe baza cheilor, folosind funcţii de dispersie. Sunt prezentate şi analizate diferite variaţii ale temei de bază; ideea de bază este că dispersia reprezintă o tehnică extrem de eficientă şi practică; operaţiile de bază pentru dicţionare necesită, în medie, doar un timp $O(1)$.

12.1. Tabele cu adresare directă

Adresarea directă este o tehnică simplă care funcţionează bine atunci când universul U al cheilor este rezonabil de mic. Să presupunem că o aplicaţie necesită o mulţime dinamică în care fiecare element are o cheie aleasă dintr-un univers $U = \{0, 1, ..., m-1\}$, unde m nu este foarte mare. Vom presupune că nu există două elemente având aceeaşi cheie.

Pentru a reprezenta mulţimea dinamică folosim un tablou sau o ***tabelă cu adresare directă*** $T[0..m-1]$, în care fiecare poziţie sau ***locaţie*** corespunde unei chei din universul U. Figura 12.1 ilustrează această abordare; locaţia k referă elementul având cheia k din mulţime. Dacă mulţimea nu conţine un element cu cheia k, atunci $T[k] =$ NIL.

Operaţiile pentru dicţionare sunt uşor de implementat.

ADRESARE-DIRECTĂ-CAUTĂ(T, k)
  returnează $T[k]$

ADRESARE-DIRECTĂ-INSEREAZĂ(T, x)
  $T[cheie[x]] \leftarrow x$

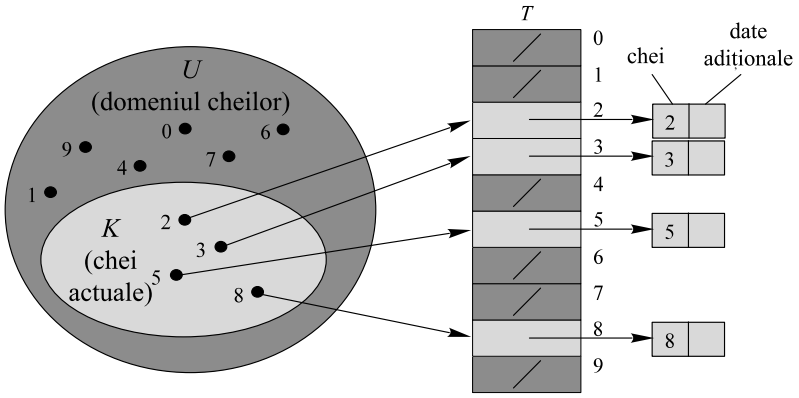


Figura 12.1 Implementarea unei mulţimi dinamice printr-un tablou cu adresare directă T. Fiecare cheie din universul $U = \{0, 1, ..., 9\}$ corespunde unui indice în tablou. Mulţimea $K = \{2, 3, 5, 8\}$ a cheilor efective determină locaţiile din tablou care conţin pointeri către elemente. Celelalte locaţii, haşurate mai întunecat, conţin NIL.

ADRESARE-DIRECTĂ-ŞTERGE(T, x)
  $T[cheie[x]] \leftarrow$ NIL

Fiecare dintre aceste operaţii este rapidă: este necesar doar un timp $O(1)$.

Pentru unele aplicaţii, elementele din mulţimea dinamică pot fi memorate chiar în tabela cu adresare directă. Aceasta înseamnă că în loc să memorăm cheia elementului şi datele adiţionale într-un obiect extern tabelei cu adresare directă, printr-un pointer dintr-o poziţie din tabelă către obiect, putem să memorăm obiectul în locaţia respectivă, economisind astfel spaţiu. Mai mult, deseori nu este necesar să memorăm câmpul cheie al obiectului, deoarece dacă avem indicele unui obiect în tabelă avem şi cheia sa. Oricum, dacă cheile nu sunt memorate, trebuie să existe o modalitate de a afirma că locaţia este goală.

Exerciţii

12.1-1 Se consideră o mulţime dinamică S care este reprezentată printr-o tabelă T cu adresare directă, de lungime m. Descrieţi o procedură care găseşte elementul maxim din S. Care este performanţa procedurii pentru cazul cel mai defavorabil?

12.1-2 Un ***vector de biţi*** este un tablou simplu de biţi (cu 0 şi 1). Un vector de biţi de lungime m ocupă mult mai puţin spaţiu decât un tablou de m pointeri. Descrieţi cum se poate folosi un vector de biţi pentru a reprezenta o mulţime dinamică având elemente distincte, fără date adiţionale. Operaţiile pentru dicţionare ar trebui să funcţioneze într-un timp $O(1)$.

12.1-3 Sugeraţi cum se poate implementa o tabelă cu adresare directă în care cheile elementelor memorate nu sunt neapărat distincte şi elementele pot avea date adiţionale. Toate cele trei operaţii pentru dicţionare (INSEREAZĂ, ŞTERGE şi CAUTĂ) ar trebui să se execute într-un timp $O(1)$. (Nu uitaţi că ŞTERGE are ca argument un pointer la obiectul ce va fi şters şi nu o cheie.)

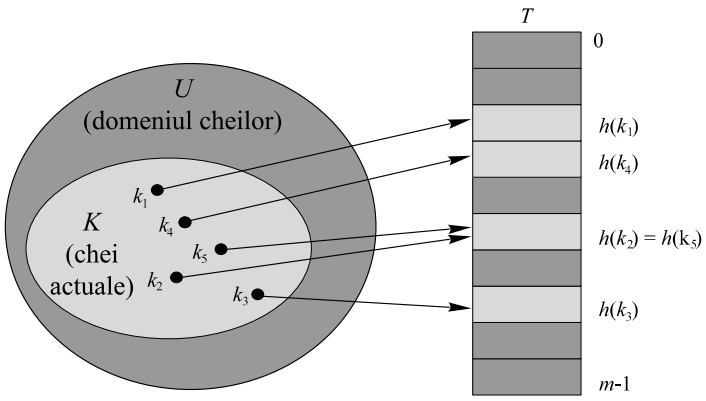


Figura 12.2 Folosirea unei funcţii de dispersie h pentru a transforma chei în poziţii din tabela de dispersie. Cheile k_2 şi k_5 se transformă în aceeaşi poziţie, deci sunt în coliziune.

12.1-4 $\star$ Dorim să implementăm un dicţionar folosind adresarea directă pe un tablou *uriaş*. La început, intrările în tablou pot conţine date nesemnificative, iar iniţializarea întregului tablou este nepractică datorită mărimii sale. Descrieţi o schemă pentru implementarea unui dicţionar cu adresare directă printr-un tablou uriaş. Fiecare obiect memorat ar trebui să utilizeze un spaţiu $O(1)$; operaţiile CAUTĂ, INSEREAZĂ şi ŞTERGE ar trebui să funcţioneze fiecare într-un timp $O(1)$; iniţializarea structurii de date ar trebui să necesite un timp $O(1)$. (*Indicaţie:* Folosiţi o stivă suplimentară, a cărei mărime să fie numărul de chei efectiv memorate în dicţionar, pentru a determina dacă o intrare dată în tabloul uriaş este validă sau nu.)

12.2. Tabele de dispersie

Dificultatea în adresarea directă este evidentă: dacă universul U este mare, memorarea tabelului T poate fi nepractică, sau chiar imposibilă, dată fiind memoria disponibilă a unui calculator uzual. Mai mult, mulţimea K a cheilor *efectiv memorate* poate fi atât de mică relativ la U, încât majoritatea spaţiului alocat pentru T ar fi irosit.

Când mulţimea K a cheilor memorate într-un dicţionar este mult mai mică decât universul U al cheilor posibile, o tabelă de dispersie necesită un spaţiu de memorie mult mai mic decât o tabelă cu adresare directă. Mai exact, cerinţele de memorare pot fi reduse la $\Theta(|K|)$, chiar şi atunci când căutarea unui element în tabela de dispersie necesită tot un timp $O(1)$. (Singurul punct slab este că această margine este stabilită pentru *timpul mediu*, în timp ce pentru adresarea directă ea era valabilă pentru *cazul cel mai defavorabil*.)

Prin adresare directă, un element având cheia k este memorat în locaţia k. Prin dispersie, acest element este memorat în locaţia $h(k)$; aceasta înseamnă că o ***funcţie de dispersie*** h este folosită pentru a calcula locaţia pe baza cheii k. În acest caz, h transformă universul U al cheilor în locaţii ale unei ***tabele de dispersie*** $T[0..m-1]$:

$$h : U \to \{0, 1, ..., m-1\}.$$

Vom spune că un element cu cheia k ***se dispersează*** în locaţia $h(k)$; de asemenea vom spune că

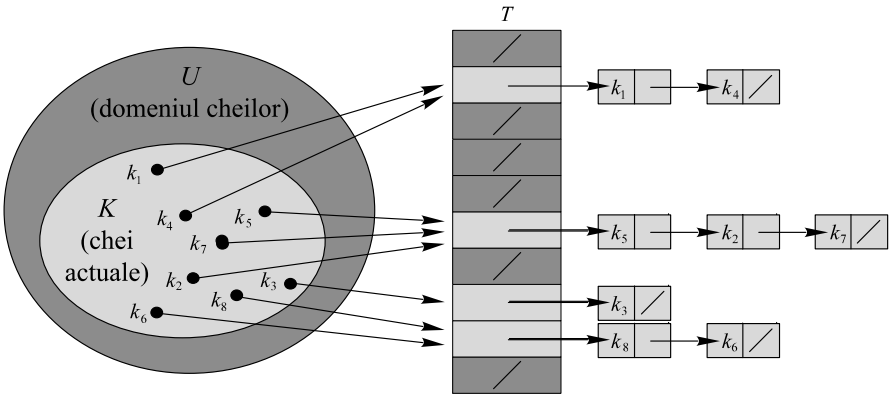


Figura 12.3 Rezolvarea coliziunii prin înlănţuire. Fiecare locaţie din tabela de dispersie $T[j]$ conţine o listă înlăţuită a tuturor cheilor a căror valoare de dispersie este j. De exemplu, $h(k_1) = h(k_4)$ şi $h(k_5) = h(k_2) = h(k_7)$.

$h(k)$ este ***valoarea de dispersie*** a cheii k. Figura 12.2 ilustrează ideea de bază. Scopul funcţiilor de dispersie este de a reduce domeniul indicilor tabloului care trebuie manipulaţi. În loc de $|U|$ valori, va trebui să manipulăm doar m valori. Cerinţele de memorare sunt reduse corespunzător.

Dezavantajul acestei idei frumoase este că două chei se pot dispersa în aceeaşi locaţie, adică se produce o ***coliziune***. Din fericire, există tehnici eficiente pentru rezolvarea conflictelor create de coliziuni.

Desigur, soluţia ideală ar fi să nu existe coliziuni. Putem încerca să atingem acest scop printr-o alegere potrivită a funcţiei de dispersie h. O idee este de a-l face pe h să pară "aleator", evitând astfel coliziunile sau cel puţin minimizând numărul lor. Termenul de "dispersie", care evocă imaginea unei fărâmiţări şi amestecări aleatoare, captează spiritul acestei abordări. (Bineînţeles, o funcţie de dispersie h trebuie să fie deterministă, în sensul că o intrare dată k trebuie să producă întotdeauna aceeaşi ieşire $h(k)$.) Deoarece $|U| > m$, există cu siguranţă două chei care să aibe aceeaşi valoare de dispersie; de aceea, evitarea totală a coliziunilor este imposibilă. Prin urmare, deşi o funcţie de dispersie "aleatoare", bine proiectată, poate minimiza numărul coliziunilor, avem în continuare nevoie de o metodă pentru rezolvarea coliziunilor ce apar.

Restul acestei secţiuni prezintă cea mai simplă tehnică de rezolvare a coliziunilor, numită înlănţuire. În secţiunea 12.4 se introduce o metodă alternativă de rezolvare a coliziunilor, numită adresare deschisă.

Rezolvarea coliziunii prin înlănţuire

Prin ***înlănţuire*** punem toate elementele ce se dispersează în aceeaşi locaţie, într-o listă înlănţuită, după cum se arată în figura 12.3. Locaţia j conţine un pointer către capul listei tuturor elementelor care se dispersează în locaţia j; dacă nu există astfel de elemente, locaţia j conţine NIL.

Operaţiile pentru dicţionare sunt uşor de implementat pe o tabelă de dispersie, în cazul în care coliziunile sunt rezolvate prin înlănţuire.

DISPERSIE-CU-INLĂNŢUIRE-INSEREAZĂ(T, x)
inserează x în capul listei $T[h(cheie[x])]$

DISPERSIE-CU-INLĂNŢUIRE-CAUTĂ(T, k)
caută un element cu cheia k în lista $T[h(k)]$

DISPERSIE-CU-INLĂNŢUIRE-ŞTERGE(T, x) indexDispersie-Cu-Inlantuire-Sterge@DISPERSIE-CU-INLĂNŢUIRE-ŞTERGE
şterge x din lista $T[h(cheie[x])]$

Timpul de execuţie pentru inserare în cazul cel mai defavorabil este $O(1)$. Pentru căutare, timpul de execuţie în cazul cel mai defavorabil este proporţional cu lungimea listei; vom analiza îndeaproape această situaţie în cele ce urmează. Ştergerea unui element x poate fi realizată într-un timp $O(1)$ dacă listele sunt dublu înlănţuite. (Dacă listele sunt simplu înlănţuite, atunci trebuie întâi să-l găsim pe x în lista $T[h(cheie[x])]$, astfel încât legătura *next* a predecesorului lui x să fie modificată corespunzător ca să-l "ocolească" pe x; în acest caz, ştergerea şi căutarea au în esenţă acelaşi timp de execuţie.)

Analiza dispersiei cu înlănţuire

Cât de bine funcţionează dispersia prin înlănţuire? În particular, cât durează căutarea unui element având o cheie dată?

Fiind dată o tabelă de dispersie T cu m locaţii ce memorează n elemente, vom defini ***factorul de încărcare*** α pentru T prin n/m, raport care reprezintă numărul mediu de elemente memorate într-o înlănţuire. Analiza noastră se bazează pe α; adică, ne imaginăm α ca fiind fix în timp ce n şi m tind la infinit. (Se observă că α poate fi mai mic, egal sau mai mare decât 1.)

Comportamentul, în cazul cel mai defavorabil, al dispersiei prin înlănţuire este slabă: toate cele n chei se dispersează în aceeaşi locaţie, creând o listă de lungime n. Timpul de căutare pentru cazul cel mai defavorabil este astfel $\Theta(n)$ plus timpul pentru calculul funcţiei de dispersie–cu nimic mai bun decât în cazul în care am fi folosit o listă înlănţuită a tuturor elementelor. Este clar că tabelele de dispersie nu sunt folosite pentru performanţa lor în cazul cel mai defavorabil.

Performanţa dispersiei în cazul mediu depinde de cât de bine distribuie (în medie) funcţia de dispersie h mulţimea cheilor ce trebuie memorate în cele m locaţii. În secţiunea 12.3 se discută aceste probleme, dar deocamdată vom presupune că orice element se poate dispersa în oricare din cele m locaţii cu aceeaşi probabilitate, independent de locul în care s-au dispersat celelalte elemente. Vom numi această ipoteză de lucru ***dispersie uniformă simplă***.

Presupunem că valoarea de dispersie $h(k)$ poate fi calculată într-un timp $O(1)$, astfel încât timpul necesar pentru a căuta un element având cheia k depinde liniar de lungimea listei $T[h(k)]$. Lăsând de o parte timpul $O(1)$ necesar pentru calculul funcţiei de dispersie şi pentru accesul la locaţia $h(k)$, să luăm în considerare numărul mediu de elemente examinate de algoritmul de căutare, adică, numărul de elemente din lista $T[h(k)]$ care sunt verificate pentru a vedea dacă cheia lor este egală cu k. Vom considera două cazuri. În primul caz, căutarea este fără succes: nici un element din tabelă nu are cheia k. În al doilea caz, căutarea găseşte cu succes un element având cheia k.

Teorema 12.1 Într-o tabelă de dispersie în care coliziunile sunt rezolvate prin înlănţuire, o căutare fără succes necesită, în medie, un timp $\Theta(1+\alpha)$, în ipoteza dispersiei uniforme simple.

Demonstraţie. În ipoteza dispersiei uniforme simple, orice cheie k se poate dispersa cu aceeaşi probabilitate în oricare din cele m locaţii. Astfel, timpul mediu pentru o căutare fără succes, pentru o cheie k, este timpul mediu pentru căutarea până la coada uneia din cele m liste. Deci, numărul mediu de elemente examinate într-o căutare fără succes este α şi timpul total necesar (incluzând timpul pentru calculul lui $h(k)$) este $\Theta(1+\alpha)$. ∎

Teorema 12.2 Într-o tabelă de dispersie în care coliziunile sunt rezolvate prin înlănţuire, o căutare cu succes necesită, în medie, un timp $\Theta(1+\alpha)$, în ipoteza dispersiei uniforme simple.

Demonstraţie. Presupunem că cheia care este căutată poate fi, cu aceeaşi probabilitate, oricare din cele n chei memorate în tabelă. De asemenea, presupunem că procedura DISPERSIE-CU-INLĂNŢUIRE-INSEREAZĂ inserează un nou element la sfârşitul unei liste şi nu la începutul ei. (Conform exerciţiului 12.2-3, timpul mediu pentru o căutare cu succes este acelaşi indiferent dacă noile elemente se inserează la începutul sau la sfârşitul listei.) Numărul mediu de elemente examinate într-o căutare cu succes este cu 1 mai mare decât numărul de elemente examinate în cazul în care elementul căutat a fost inserat (deoarece fiecare nou element se adaugă la sfârşitul listei). În concluzie, pentru a afla numărul mediu de elemente examinate, vom considera media celor n obiecte din tabelă, ca fiind 1 plus lungimea presupusă a listei în care se adaugă al i-lea element. Lungimea medie a listei respective este $(i-1)/m$ şi deci, numărul mediu de elemente examinate într-o căutare cu succes este:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) = 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right) = 1+\frac{\alpha}{2}-\frac{1}{2m}\,.$$

Deci, timpul total necesar unei căutări cu succes (incluzând timpul pentru calculul funcţiei de dispersie) este $\Theta(2+\alpha/2-1/2m) = \Theta(1+\alpha)$. ∎

Ce semnificaţie are această analiză? Dacă numărul de locaţii din tabela de dispersie este cel puţin proporţional cu numărul de elemente din tabelă, avem $n = O(m)$ şi, prin urmare, $\alpha = n/m = O(m)/m = O(1)$. Deci, căutarea necesită, în medie, un timp constant. Deoarece inserarea necesită un timp $O(1)$, în cazul cel mai defavorabil (vezi exerciţiul 12.2-3) iar ştergerea necesită un timp $O(1)$ în cazul cel mai defavorabil, când listele sunt dublu înlănţuite, toate operaţiile pentru dicţionare pot fi efectuate în medie într-un timp $O(1)$.

Exerciţii

12.2-1 Presupunând că folosim o funcţie de dispersie aleatoare h pentru a dispersa n chei distincte într-un tablou T de dimensiune m, care este numărul mediu de coliziuni? Mai exact, care este cardinalul probabil al mulţimii $\{(x,y) : h(x) = h(y)\}$?

12.2-2 Ilustraţi inserarea cheilor 5, 28, 19, 15, 20, 33, 12, 17, 10 într-o tabelă de dispersie cu coliziunile rezolvate prin înlănţuire. Tabela are 9 locaţii, iar funcţia de dispersie este $h(k) = k \bmod 9$.

12.2-3 Argumentaţi că timpul mediu pentru o căutare cu succes prin înlănţuire este acelaşi indiferent dacă noile elemente se inserează la începutul, respectiv la sfârşitul unei liste. (*Indicaţie:* Arătaţi că timpul mediu pentru o căutare cu succes este acelaşi pentru *oricare* două ordine ale unei liste.)

12.2-4 Profesorul Marley emite ipoteza că se poate îmbunătăţi esenţial performanţa dacă modificăm schema de înlănţuire astfel încât fiecare listă să fie păstrată ordonată. În ce mod afectează modificarea profesorului timpul de execuţie pentru căutari cu succes, căutări fără succes, inserări şi ştergeri?

12.2-5 Sugeraţi cum poate fi alocat şi dealocat spaţiul de gestionare pentru elemente în cadrul tabelei de dispersie prin legarea tuturor locaţiilor nefolosite într-o listă liberă. Presupunem că o locaţie poate memora un "indicator" (engl. *flag*) împreună cu un element şi un pointer, sau cu doi pointeri. Toate operaţiile pentru dicţionare pentru lista liberă trebuie să funcţioneze într-un timp $O(1)$. Este necesar ca lista liberă să fie dublu înlănţuită sau este suficient să fie simplu înlănţuită?

12.2-6 Arătaţi că dacă $|U| > nm$, există o submulţime a lui U de mărime n ce conţine chei care se dispersează toate în aceeaşi locaţie, astfel încât timpul de căutare, pentru dispersie cu înlănţuire, în cazul cel mai defavorabil, este $\Theta(n)$.

12.3. Funcţii de dispersie

În această secţiune vom discuta unele aspecte legate de construirea de funcţii de dispersie bune, apoi vom prezenta trei scheme pentru crearea lor: dispersia prin diviziune, dispersia prin înmulţire şi dispersia universală.

Ce înseamnă o funcţie de dispersie bună?

O funcţie de dispersie bună satisface (aproximativ) ipoteza dispersiei uniforme simple: fiecare cheie se poate dispersa cu aceeaşi probabilitate în oricare dintre cele m locaţii. Mai formal, să presupunem că fiecare cheie este aleasă independent din U conform unei distribuţii de probabilitate P; deşi, $P(k)$ este probabilitatea de a fi aleasă cheia k. Atunci ipoteza dispersiei uniforme simple constă în

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \text{ pentru } j = 0, 1, ..., m-1 \,. \tag{12.1}$$

Din păcate, în general, nu este posibilă verificarea acestei condiţii, deoarece de regulă distribuţia P nu este cunoscută.

Uneori (relativ rar) distribuţia P este cunoscută. De exemplu, presupunem că se cunoaşte faptul că cheile sunt numere reale aleatoare k, independent şi uniform distribuite în intervalul $0 \leq k < 1$. În acest caz se poate arăta că funcţia de dispersie

$$h(k) = \lfloor km \rfloor$$

satisface relaţia (12.1).

În practică, se pot folosi tehnici euristice pentru a crea funcţii de dispersie care par a se comporta bine. Informaţia calitativă despre P este uneori utilă în procesul de construcţie al lor. De exemplu, să considerăm o tabelă de simboluri a unui compilator, în care cheile sunt şiruri de caractere arbitrare, reprezentând identificatori dintr-un program. O situaţie des întâlnită este aceea în care simboluri foarte asemănătoare, ca `pt` şi `pts`, apar în acelaşi program. O funcţie de dispersie bună va minimiza posibilitatea ca asemenea variaţii să se disperseze în aceeaşi locaţie.

O abordare uzuală este de a obţine valoarea de dispersie într-un mod care se vrea independent de orice model sau şablon ce poate exista între date. De exemplu, "metoda diviziunii" (discutată în detaliu mai jos) calculează valoarea de dispersie ca fiind restul împărţirii cheii la un număr prim specificat. În afara cazului în care numărul este într-un fel dependent de şabloanele din distribuţia de probabilitate P, această metodă dă rezultate bune.

În final, să remarcăm faptul că unele aplicaţii ale funcţiilor de dispersie pot impune proprietăţi mai tari decât cele asigurate de dispersia uniformă simplă. De exemplu, am putea dori ca unele chei care sunt "apropiate" într-un anumit sens să producă valori de dispersie care să fie total diferite. (Această proprietate este dorită în special când folosim verificarea liniară, definită în secţiunea 12.4.)

Interpretarea cheilor ca numere naturale

Majoritatea funcţiilor de dispersie presupun universul cheilor din mulţimea $\mathbb{N} = \{0, 1, 2, ...\}$ a numerelor naturale. Astfel, dacă cheile nu sunt numere naturale, trebuie găsită o modalitate pentru a le interpreta ca numere naturale. De exemplu, o cheie care este un şir de caractere poate fi interpretată ca un întreg într-o bază de numeraţie aleasă convenabil. Prin urmare, identificatorul `pt` poate fi interpretat ca o pereche de numere zecimale (112,116), pentru că `p` = 112 şi `t` = 116 în mulţimea codurilor ASCII; atunci `pt` exprimat ca un întreg în baza 128 devine $(112 \cdot 128)+116 = 14452$. În mod obişnuit, în orice aplicaţie se poate crea direct o astfel de metodă simplă de a interpreta fiecare cheie ca un număr natural (posibil mare). În continuare, vom presupune că avem chei numere naturale.

12.3.1. Metoda diviziunii

Prin ***metoda diviziunii*** pentru crearea funcţiilor de dispersie, transformăm o cheie k într-una din cele m locaţii considerând restul împărţirii lui k la m. Prin urmare, funcţia de dispersie este

$$h(k) = k \mod m\,.$$

De exemplu, dacă tabela de dispersie are dimensiunea $m = 12$ şi cheia este $k = 100$, atunci $h(k) = 4$. Deoarece necesită doar o singură operaţie de împărţire, dispersia prin diviziune este rapidă.

Când folosim metoda diviziunii, de obicei evităm anumite valori ale lui m. De exemplu, m nu ar trebui să fie o putere a lui 2, pentru că dacă $m = 2^p$, atunci $h(k)$ reprezintă doar primii p biţi ai lui k. În afara cazului în care se cunoaşte apriori că distribuţia de probabilitate pe chei produce cu aceeaşi probabilitate oricare dintre şabloanele primilor p biţi, este mai bine să construim funcţia de dispersie ca dependentă de toţi biţii cheii. Puterile lui 10 ar trebui evitate dacă aplicaţia lucrează cu numere zecimale ca şi chei, pentru că în acest caz funcţia de dispersie

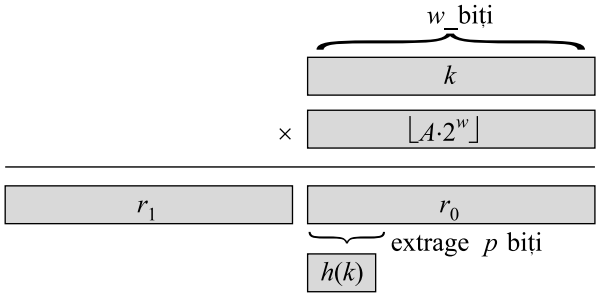


Figura 12.4 Metoda înmulţirii pentru dispersie. Reprezentarea pe w biţi a cheii k este înmulţită cu valoarea pe w biţi $\lfloor A \cdot 2^w \rfloor$, unde $0 < A < 1$ este o constantă aleasă convenabil. Primii p biţi ai celei de a doua jumătăţi a produsului, de lungime w biţi, formează valoarea de dispersie $h(k)$ dorită.

nu depinde de toate cifrele zecimale ale lui k. În final, se poate arăta că dacă $m = 2^p - 1$ şi k este un şir de caractere interpretat în baza 2^p, două şiruri de caractere care sunt identice, exceptând o transpoziţie a două caractere adiacente, se vor dispersa în aceeaşi valoare.

Valori bune pentru m sunt numerele prime nu prea apropiate de puterile exacte ale lui 2. De exemplu, să presupunem că dorim să alocăm o tabelă de dispersie, cu coliziunile rezolvate prin înlănţuire, pentru a reţine aproximativ $n = 2000$ şiruri de caractere, unde un caracter are 8 biţi. Nu ne deranjează să examinăm în medie 3 elemente într-o căutare fără succes, deci vom aloca o tabelă de dispersie de dimensiune $m = 701$. Numărul 701 este ales pentru că este un număr prim apropiat de $2000/3$, dar nu este apropiat de nici o putere a lui 2. Tratând fiecare cheie k ca un întreg, funcţia noastră de dispersie va fi

$$h(k) = k \mod 701 .$$

Ca o măsură de precauţie, putem să verificăm cât de uniform distribuie această funcţie de dispersie mulţimi de chei în locaţii, unde cheile sunt alese din date "reale".

12.3.2. Metoda înmulţirii

Metoda înmulţirii pentru crearea de funcţii de dispersie operează în doi paşi. În primul pas, înmulţim cheia k cu o constantă A din intervalul $0 < A < 1$ şi extragem partea fracţionară a lui kA. Apoi, înmulţim această valoare cu m şi considerăm partea întreagă inferioară a rezultatului. Pe scurt, funcţia de dispersie este

$$h(k) = \lfloor m(kA \mod 1) \rfloor ,$$

unde "$kA \mod 1$" înseamnă partea fracţionară a lui kA, adică, $kA - \lfloor kA \rfloor$.

Un avantaj al metodei înmulţirii este că valoarea lui m nu este critică. De obicei o alegem ca fiind o putere a lui 2 – $m = 2^p$ pentru un întreg p – pentru că atunci putem implementa funcţia pe majoritatea calculatoarelor după cum urmează. Să presupunem că lungimea cuvântului maşinii este de w biţi şi că k încape pe un singur cuvânt. Referindu-ne la figura 12.4, înmulţim întâi k cu întregul pe w biţi $\lfloor A \cdot 2^w \rfloor$. Rezultatul este o valoare pe $2w$ biţi $r_1 2^w + r_0$, unde r_1 este primul cuvânt al produsului şi r_0 este al doilea cuvânt al produsului. Valoarea de dispersie de p biţi dorită constă din cei mai semnificativi p biţi ai lui r_0.

Deşi această metodă funcţionează cu orice valoare a constantei A, ea lucrează mai bine cu anumite valori decât cu altele. Alegerea optimă depinde de caracteristicile datelor care sunt dispersate. Knuth [123] discută alegerea lui A în detaliu şi sugerează că

$$A = (\sqrt{5} - 1)/2 \approx 0.6180339887... \tag{12.2}$$

se va comporta relativ bine.

De exemplu, dacă avem $k = 123456, m = 10000$ şi A din relaţia (12.2), atunci

$$\begin{aligned} h(k) &= \lfloor 10000 \cdot (123456 \cdot 0.61803... \bmod 1) \rfloor = \lfloor 10000 \cdot (76300.0041151... \bmod 1) \rfloor = \\ &= \lfloor 10000 \cdot 0.0041151... \rfloor = \lfloor 41.151... \rfloor = 41 \, . \end{aligned}$$

12.3.3. Dispersia universală

Dacă cheile ce se dispersează sunt alese de un adversar maliţios, atunci el poate alege n chei care toate se vor dispersa în aceeaşi locaţie, rezultând un timp mediu de acces $\Theta(n)$. În cazul cel mai defavorabil, orice funcţie de dispersie fixată este vulnerabilă la acest tip de comportament; singura modalitate eficientă de a îmbunătăţi această situaţie este de a alege funcţia de dispersie în mod *aleator* astfel încât alegerea să fie *independentă* de cheile care se vor memora. Această abordare, numită ***dispersie universală***, produce în medie o performanţă bună, indiferent de ce chei sunt alese de către adversar.

Ideea care stă la baza dispersiei universale este de a selecta funcţia de dispersie în mod aleator în momentul execuţiei dintr-o clasă de funcţii construită cu atenţie. Ca şi în cazul sortării rapide, randomizarea asigură că nici o intrare nu va produce comportamentul în cazul cel mai defavorabil. Datorită randomizării, algoritmul se poate comporta diferit la fiecare execuţie, chiar pentru o aceeaşi intrare. Această abordare asigură o performanţă bună pentru cazul mediu, indiferent de cheile care sunt date ca intrare. Revenind la exemplul cu tabela de simboluri a unui compilator, descoperim că alegerea identificatorilor de către programator nu poate cauza, din punct de vedere al consistenţei, performanţe slabe la dispersie. Performanţa slabă apare doar atunci când compilatorul alege o funcţie de dispersie aleatoare care provoacă o dispersie slabă a mulţimii identificatorilor, dar probabilitatea ca această situaţie să apară este mică şi este aceeaşi pentru orice mulţime de identificatori de aceeaşi mărime.

Fie $\mathcal{H}$ o colecţie finită de funcţii de dispersie care transformă un univers dat U al cheilor, în domeniul $\{0, 1, ..., m-1\}$. O astfel de colecţie se numeşte ***universală*** dacă pentru fiecare pereche de chei distincte $x, y \in U$, numărul de funcţii de dispersie $h \in \mathcal{H}$ pentru care $h(x) = h(y)$ este exact $|\mathcal{H}|/m$. Cu alte cuvinte, cu o funcţie de dispersie aleasă aleator din $\mathcal{H}$, şansa unei coliziuni între x şi y când $x \neq y$ este exact $1/m$, care este exact şansa unei coliziuni dacă $h(x)$ şi $h(y)$ sunt alese aleator din mulţimea $\{0, 1, ..., m-1\}$.

Următoarea teoremă arată că o clasă universală de funcţii de dispersie dă un comportament bun în cazul mediu.

Teorema 12.3 Dacă h este aleasă dintr-o colecţie universală de funcţii de dispersie şi este folosită pentru a dispersa n chei într-o tabelă de dimensiune m, unde $n \leq m$, numărul mediu de coliziuni în care este implicată o cheie particulară x, este mai mic decât 1.

Demonstraţie. Pentru fiecare pereche y, z de chei distincte, fie c_{yz} variabila aleatoare care are valoarea 1 dacă $h(y) = h(z)$ (adică, dacă y şi z sunt în coliziune folosind h) şi 0 în caz contrar.

Deoarece, prin definiţie, o singură pereche de chei sunt în coliziune cu probabilitatea $1/m$, avem

$$E[c_{yz}] = 1/m \, .$$

Fie C_x numărul total de coliziuni în care este implicată cheia x dintr-o tabelă de dispersie T de dimensiune m conţinând n chei. Din relaţia (6.24) obţinem

$$E[C_x] = \sum_{\substack{y \in T \\ y \neq x}} E[c_{xy}] = \frac{n-1}{m} \, .$$

Deoarece $n \leq m$, obţinem $E[C_x] < 1$. ∎

Dar cât de uşor este să construim o clasă universală de funcţii de dispersie? Este relativ uşor, după cum se poate demonstra folosind unele cunoştinţe din teoria numerelor. Să alegem dimensiunea m a tabelei noastre ca fiind număr prim (ca şi în metoda diviziunii). Descompunem o cheie x în $r+1$ octeţi (de exemplu, caractere sau subşiruri binare de caractere de lungime fixă), astfel încât $x = \langle x_0, x_1, ...x_r \rangle$; singura cerinţă este ca valoarea maximă a unui octet să fie mai mică decât m. Notăm prin $a = \langle a_0, a_1, ...a_r \rangle$ un şir de $r+1$ elemente alese aleator din mulţimea $\{0, 1, ..., m-1\}$. Definim o funcţie de dispersie corespunzătoare $h_a \in \mathcal{H}$ astfel:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \mod m \, . \tag{12.3}$$

Cu această definiţie,

$$\mathcal{H} = \bigcup_a \{h_a\}. \tag{12.4}$$

are m^{r+1} elemente.

Teorema 12.4 Clasa $\mathcal{H}$ definită de relaţiile (12.3) şi (12.4) este o clasă universală de funcţii de dispersie.

Demonstraţie. Considerăm o pereche oarecare de chei distincte x şi y. Presupunem că $x_0 \neq y_0$. (O argumentaţie similară se poate face pentru diferenţa dintre oricare alte poziţii ale octeţilor.) Pentru orice valori fixe ale lui $a_1, a_2, ..., a_r$ există exact o valoare a lui a_0 care satisface ecuaţia $h(x) = h(y)$; acest a_0 este soluţia ecuaţiei

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m} \, .$$

Pentru a observa această proprietate să remarcăm faptul că deoarece m este prim, cantitatea nenulă $x_0 - y_0$ are un invers faţă de înmulţirea modulo m şi astfel există o soluţie unică pentru a_0 modulo m. (Vezi secţiunea 33.4.) Prin urmare, fiecare pereche de chei x şi y este în coliziune pentru exact m^r valori ale lui a, deoarece ea este în coliziune exact o dată pentru fiecare valoare posibilă a lui $\langle a_1, a_2, ..., a_r \rangle$ (de exemplu, pentru valoarea unică a lui a_0 observată mai sus). Deoarece există m^{r+1} valori posibile pentru secvenţa a, cheile x şi y sunt în coliziune exact cu probabilitatea $m^r/m^{r+1} = 1/m$. În concluzie, $\mathcal{H}$ este universală. ∎

Exerciţii

12.3-1 Presupunem că dorim să efectuăm o căutare într-o listă înlănţuită de lungime n, în care fiecare element conţine o cheie k împreună cu o valoare de dispersie $h(k)$. Fiecare cheie este un şir lung de caractere. Cum putem profita de valorile de dispersie când căutăm în listă un element cu o cheie dată?

12.3-2 Presupunem că un şir de r caractere este dispersat pe m poziţii, fiind considerat ca număr în baza 128, căruia i se aplică metoda diviziunii. Numărul m este uşor de reprezentat în memorie ca un cuvânt pe 32 de biţi, dar şirul de r caractere, tratat ca un număr în baza 128, necesită multe cuvinte. Cum putem aplica metoda diviziunii pentru a calcula valoarea de dispersie a şirului de caractere fără a folosi mai mult decât un număr suplimentar constant de cuvinte de memorie în afară de şirul propriu-zis?

12.3-3 Se consideră o versiune a metodei diviziunii în care $h(k) = k \mod m$, unde $m = 2^p - 1$ şi k este un şir de caractere interpretat în baza 2^p. Arătaţi că dacă şirul de caractere x poate fi obţinut din şirul de caractere y prin permutări de caractere, atunci x şi y se dispersează în aceeaşi valoare. Daţi un exemplu de aplicaţie în care această proprietate nu este dorită pentru o funcţie de dispersie.

12.3-4 Se consideră o tabelă de dispersie de dimensiune $m = 1000$ şi funcţia de dispersie $h(k) = \lfloor m(kA \mod 1) \rfloor$ pentru $A = (\sqrt{5} - 1)/2$. Calculaţi locaţiile în care se pun cheile 61, 62, 63, 64, şi 65.

12.3-5 Arătaţi că dacă punem restricţia ca fiecare componentă a_i din a din relaţia (12.3) să fie nenulă, atunci mulţimea $\mathcal{H} = \{h_a\}$ definită ca în relaţia (12.4) nu este universală. (*Indicaţie:* Se consideră cheile $x = 0$ şi $y = 1$.)

12.4. Adresarea deschisă

Prin ***adresare deschisă***, toate elementele sunt memorate în interiorul tabelei de dispersie. Prin urmare, fiecare intrare în tabelă conţine fie un element al mulţimii dinamice, fie NIL. Când căutăm un element, vom examina sistematic locaţiile tabelei fie până când este găsit elementul dorit, fie până când este clar că elementul nu este în tabelă. Nu există liste sau elemente memorate în afara tabelei, aşa cum se întâmplă în cazul înlănţuirii. Prin urmare, prin adresare deschisă, tabela de dispersie se poate "umple" astfel încât nici o inserare nu mai este posibilă; factorul de încărcare α nu poate depăşi niciodată valoarea 1.

Desigur, am putea memora listele înlănţuite pentru a crea legăturile în cadrul tabelei de dispersie, în locaţiile altfel neutilizate ale acesteia (vezi exerciţiul 12.2-5), dar avantajul adresării deschise este că evită total folosirea pointerilor. Secvenţa de locaţii care se examinează nu se determină folosind pointerii, ci se *calculează*. Spaţiul de memorie suplimentar, eliberat prin faptul că nu memorăm pointerii, oferă tabelei de dispersie un număr mai mare de locaţii pentru acelaşi spaţiu de memorie, putând rezulta coliziuni mai puţine şi acces mai rapid.

Pentru a realiza inserarea folosind adresarea deschisă, examinăm succesiv sau ***verificăm*** tabela de dispersie până când găsim o locaţie liberă în care să punem cheia. În loc să fie fixat în ordinea $0, 1, ..., m - 1$ (care necesită un timp de căutare $\Theta(n)$), şirul de poziţii examinate

depinde de cheia ce se inserează. Pentru a determina ce locaţii sunt verificate, extindem funcţia de dispersie astfel încât ea să includă şi numărul de verificare (începând de la 0) ca un al doilea argument. Astfel, funcţia de dispersie devine

$$h : U \times \{0, 1, ..., m-1\} \to \{0, 1, ..., m-1\} .$$

În cadrul adresării deschise, cerem ca pentru fiecare cheie k, ***secvenţa de verificare***

$$\langle h(k, 0), h(k, 1), ..., h(k, m-1)\rangle$$

să fie o permutare a lui $\langle 0, 1, ..., m-1\rangle$, astfel încât fiecare poziţie din tabela de dispersie să fie considerată ca o eventuală locaţie pentru o nouă cheie pe măsură ce se umple tabela. În algoritmul următor, presupunem că elementele din tabela de dispersie T sunt chei, fără informaţii adiţionale; cheia k este identică cu elementul care conţine cheia k. Fiecare locaţie conţine fie o cheie, fie NIL (dacă locaţia este liberă).

DISPERSIE-INSEREAZĂ(T, k)
1: $i \leftarrow 0$
2: **repetă**
3: $\quad j \leftarrow h(k, i)$
4: $\quad$ **dacă** $T[j] =$ NIL **atunci**
5: $\quad\quad T[j] \leftarrow k$
6: $\quad\quad$ **returnează** j
7: $\quad$ **altfel**
8: $\quad\quad i \leftarrow i + 1$
9: **până când** $i = m$
10: **eroare** "depăşire tabela de dispersie"

Algoritmul pentru căutarea unei chei k examinează aceeaşi secvenţă de locaţii pe care o foloseşte şi algoritmul de inserare atunci când s-a inserat cheia k. Prin urmare, căutarea se poate termina (fără succes) când se întâlneşte o locaţie liberă, pentru că k ar fi fost inserat în acea locaţie şi nu mai încolo în secvenţa de verificare. (Se observă că această argumentaţie presupune că o dată introduse, cheile nu mai sunt şterse din tabela de dispersie.) Procedura DISPERSIE-CAUTĂ are ca intrare o tabelă de dispersie T şi o cheie k şi returnează j dacă locaţia j conţine cheia k sau NIL dacă cheia k nu există în tabela T.

DISPERSIE-CAUTĂ(T, k)
1: $i \leftarrow 0$
2: **repetă**
3: $\quad j \leftarrow h(k, i)$
4: $\quad$ **dacă** $T[j] = k$ **atunci**
5: $\quad\quad$ **returnează** j
6: $\quad\quad i \leftarrow i + 1$
7: **până când** $T[j] =$ NIL **sau** $i = m$
8: **returnează** NIL

Ştergerea dintr-o tabelă de dispersie cu adresare deschisă este dificilă. Când ştergem o cheie dintr-o locaţie i, nu putem marca pur şi simplu acea locaţie ca fiind liberă memorând în ea

valoarea NIL. Procedând astfel, va fi imposibil să accesăm orice cheie k a cărei inserare a verificat locaţia i şi a găsit-o ocupată. O soluţie este de a marca locaţia, memorând în ea valoarea specială ŞTERS în loc de NIL. Apoi vom modifica procedura DISPERSIE-INSEREAZĂ astfel încât să trateze astfel de locaţii ca şi când ar fi libere, astfel încât o nouă cheie să poată fi inserată. Nu este necesară nici o modificare în DISPERSIE-CAUTĂ, deoarece va trece peste valorile ŞTERS în timpul căutării. Procedând astfel, timpii de căutare nu mai depind de factorul de încărcare α şi din acest motiv în cazul în care cheile trebuie şterse, se preferă, în mod uzual, înlănţuirea ca tehnică de rezolvare a coliziunilor.

În analiza noastră, folosim ipoteza ***dispersiei uniforme:*** presupunem că fiecare cheie considerată poate avea, cu aceeaşi probabilitate, oricare dintre cele $m!$ permutări ale mulţimii $\{0, 1, ..., m-1\}$ ca secvenţă de verificare. Dispersia uniformă generalizează noţiunea de dispersie uniformă simplă, definită anterior, pentru situaţia în care funcţia de dispersie produce nu doar un singur număr, ci o întreagă secvenţă de verificare. Oricum, dispersia uniformă reală este dificil de implementat şi în practică sunt folosite aproximări convenabile (ca dispersia dublă, definită mai târziu).

În mod obişnuit sunt utilizate trei tehnici pentru a calcula secvenţele de verificare necesare adresării deschise: verificarea liniară, verificarea pătratică şi dispersia dublă. Toate aceste tehnici garantează că $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$ este o permutare a lui $\langle 0, 1, ..., m-1\rangle$ pentru fiecare cheie k. Nici una din aceste tehnici nu satisface condiţia dispersiei uniforme, pentru că nici una dintre ele nu e capabilă să genereze mai mult de m^2 secvenţe de verificare diferite (în loc de $m!$ cât cere dispersia uniformă). Dispersia dublă are cel mai mare număr de secvenţe de verificare şi, după cum era de aşteptat, dă cele mai bune rezultate.

Verificarea liniară

Fiind dată o funcţie de dispersie ordinară $h' : U \to \{0, 1, ..., m-1\}$, metoda ***verificării liniare*** foloseşte funcţia de dispersie

$$h(k,i) = (h'(k) + i) \mod m$$

pentru $i = 0, 1, ..., m-1$. Fiind dată o cheie k, prima locaţie examinată (verificată) este $T[h'(k)]$. Apoi examinăm locaţia $T[h'(k)+1]$ şi aşa mai departe până la locaţia $T[m-1]$. Apoi continuăm circular cu locaţiile $T[0], T[1], ...$, până când, în final, verificăm locaţia $T[h'(k)-1]$. Deoarece poziţia de start a verificării determină întreaga secvenţă de verificare liniară, prin care sunt folosite doar m secvenţe de verificări distincte.

Verificarea liniară este uşor de implementat, dar apare o problemă cunoscută sub numele de ***grupare primară***. În acest caz se formează şiruri lungi de locaţii ocupate, crescând timpul mediu de căutare. De exemplu, dacă avem $n = m/2$ chei într-o tabelă, în care fiecare locaţie de indice par este ocupată şi fiecare locaţie de indice impar este liberă, atunci căutarea fără succes necesită, 1,5 verificări în cazul mediu. Dacă primele $n = m/2$ locaţii sunt cele ocupate, numărul mediu de verificări creşte la aproximativ $n/4 = m/8$. Grupările au probabilitate mare de apariţie, pentru că dacă o locaţie liberă este precedată de i locaţii ocupate, atunci probabilitatea ca locaţia liberă să fie următoarea completată este $(i+1)/m$, comparativ cu probabilitatea de $1/m$ pentru cazul în care locaţia precedentă ar fi fost liberă. Prin urmare, şirurile de locaţii ocupate tind să se lungească şi verificarea liniară nu este o aproximare foarte bună a dispersiei uniforme.

Verificarea pătratică

Verificarea pătratică foloseşte o funcţie de dispersie de forma

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m\,, \tag{12.5}$$

unde (ca şi în cazul verificării liniare) h' este o funcţie de dispersie auxiliară, c_1 şi $c_2 \neq 0$ sunt constante auxiliare şi $i = 0, 1, ..., m-1$. Poziţia verificată iniţial este $T[h'(k)]$; următoarele poziţii examinate sunt decalate cu cantităţi ce depind într-o manieră pătratică de numărul de verificare i. Această metodă lucrează mult mai bine decât verificarea liniară, dar pentru a folosi integral tabela de dispersie valorile lui c_1, c_2 şi m trebuie determinate corespunzător. Problema 12-4 ilustrează o modalitate de determinare a acestor parametri. De asemenea, dacă două chei au aceeaşi poziţie de start a verificării, atunci secvenţele lor de verificare coincid, pentru că $h(k_1, 0 = h(k_2, 0)$ implică $h(k_1, i) = h(k_2, i)$. Această situaţie conduce la o formă mai uşoară de grupare, numită ***grupare secundară***. Ca şi în cazul verificării liniare, verificarea iniţială determină întreaga secvenţă, deci sunt folosite doar m secvenţe de verificare distincte.

Dispersia dublă

dispersia dublă este una dintre cele mai bune metode disponibile pentru adresarea deschisă, deoarece permutările produse au multe din caracteristicile permutărilor alese aleator. ***dispersia dublă*** foloseşte o funcţie de dispersie de forma

$$h(k,i) = (h_1(k) + i h_2(k)) \mod m\,,$$

unde h_1 şi h_2 sunt funcţii de dispersie auxiliare. Poziţia examinată iniţial este $T[h_1(k)]$; poziţiile succesive de verificare sunt decalate faţă de poziţiile anterioare cu $h_2(k)$ modulo m. Astfel, contrar situaţiei de verificarea liniară sau pătratică, în acest caz secvenţa de verificare depinde în două moduri de cheia k, pentru că fie poziţia iniţială de verificare, fie decalajul, fie amândouă pot varia. Figura 12.5 ilustrează un exemplu de inserare prin dispersie dublă.

Valoarea $h_2(k)$ şi dimensiunea tabelei de dispersie m trebuie să fie prime între ele pentru a fi parcursă întreaga tabelă de dispersie. În caz contrar, dacă m şi $h_2(k)$ au un cel mai mare divizor comun $d > 1$ pentru o cheie k, atunci o căutare pentru cheia k va examina doar a $(1/d)$-a parte din tabela de dispersie. (Vezi capitolul 33.) O modalitate convenabilă de a asigura această condiţie este de a avea un număr m ca o putere a lui 2 şi de a construi h_2 astfel încât să producă întotdeauna un număr impar. O altă modalitate este de a avea m prim şi de a construi h_2 astfel încât să returneze întotdeauna un întreg pozitiv mai mic decât m. De exemplu, putem alege m prim şi

$$\begin{aligned} h_1(k) &= k \mod m\,, \\ h_2(k) &= 1 + (k \mod m')\,, \end{aligned}$$

unde m' este ales să fie un pic mai mic decât m (să zicem $m-1$ sau $m-2$). De exemplu, dacă $k = 123456, m = 701$ şi $m' = 700$, avem $h_1(k) = 80$ şi $h_2(k) = 257$, deci prima verificare se află la poziţia 80 şi fiecare a 257-a locaţie (modulo m) este examinată până când cheia este găsită sau a fost examinată toată tabela.

dispersia dublă reprezintă o îmbunătăţire faţă de verificarea liniară sau pătratică în sensul că sunt folosite $\Theta(m^2)$ secvenţe de verificare, faţă de $\Theta(m)$, pentru că fiecare pereche posibilă $(h_1(k), h_2(k))$ produce o secvenţă de verificare distinctă c si când cheia variază, poziţia iniţială

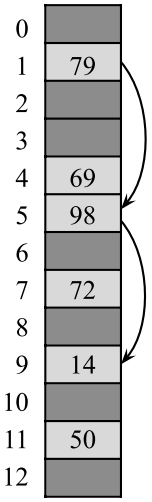


Figura 12.5 Inserarea prin dispersie dublă. Avem o tabelă de dispersie de dimensiune 13, cu $h_1(k) = k \bmod 13$ şi $h_2(k) = 1 + (k \bmod 11)$. Deoarece $14 \equiv 1 \bmod 13$ şi $14 \equiv 3 \bmod 11$, cheia 14 va fi inserată în locaţia liberă 9, după ce locaţiile 1 şi 5 au fost examinate şi găsite ca fiind deja ocupate.

a verificării $h_1(k)$ şi decalajul $h_2(k)$ pot varia independent. Ca rezultat, performanţa dispersiei duble apare ca fiind foarte apropiată de performanţa schemei "ideale" a dispersiei uniforme.

Analiza dispersiei cu adresare deschisă

Analiza noastră pentru adresarea deschisă, este, ca şi analiza pentru înlănţuire, exprimată în termenii factorului de încărcare α al tabelei de dispersie, când n şi m tind spre infinit. Să reamintim că dacă sunt memorate n elemente într-o tabelă cu m locaţii, numărul mediu de elemente pe locaţie este $\alpha = n/m$. Desigur, prin adresare deschisă, avem cel mult un element într-o locaţie şi, prin urmare, $n \leq m$, ceea ce implică $\alpha \leq 1$.

Presupunem că este folosită dispersia uniformă. În această schemă idealizată, secvenţa de verificare $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$ pentru orice cheie k poate să apară sub forma oricărei permutări a mulţimii $\langle 0, 1, ..., m-1\rangle$. Aceasta înseamnă că fiecare secvenţă de verificare posibilă poate fi folosită ca secvenţă de verificare pentru o inserare sau o căutare. Desigur, o cheie dată are o secvenţă de verificare unică, fixă, asociată ei; considerând distribuţia de probabilitate pe spaţiul cheilor şi operaţia funcţiei de dispersie pe chei, fiecare secvenţă de verificare posibilă are aceeaşi probabilitate de apariţie.

Analizăm acum numărul mediu de verificări pentru dispersia cu adresare deschisă, în ipoteza dispersiei uniforme şi începem cu o analiză a numărului de verificări ce apar într-o căutare fără succes.

Teorema 12.5 Fiind dată o tabelă de dispersie cu adresare deschisă cu factorul de încărcare $\alpha = n/m < 1$, în ipoteza dispersiei uniforme, numărul mediu de verificări dintr-o căutare fără succes este cel mult $1/(1-\alpha)$.

Demonstraţie. Într-o căutare fără succes, fiecare verificare, cu excepţia ultimei, accesează o

locaţie ocupată care nu conţine cheia dorită, iar ultima locaţie examinată este liberă. Definim

p_i = Pr { exact i verificări accesează locaţiile ocupate }

pentru $i = 0, 1, 2, ...$ Pentru $i > n$, avem $p_i = 0$, pentru că putem găsi cel mult n locaţii deja ocupate. Astfel, numărul mediu de verificări este

$$1 + \sum_{i=1}^{\infty} ip_i \tag{12.6}$$

Pentru a evalua relaţia (12.6), definim

q_i = Pr { cel puţin i verificări accesează locaţiile ocupate }

pentru $i = 0, 1, 2, ...$ Apoi putem folosi identitatea (6.28):

$$\sum_{i=1}^{\infty} ip_i = \sum_{i=1}^{\infty} q_i \, .$$

Care este valoarea lui q_i pentru $i \geq 1$? Probabilitatea ca prima verificare să acceseze o locaţie ocupată este n/m; prin urmare,

$$q_1 = \frac{n}{m} \, .$$

Prin dispersie uniformă, a doua verificare, în cazul în care este necesară, va fi la una din cele $m-1$ locaţii rămase neverificate, din care $n-1$ sunt ocupate. Facem a doua verificare doar dacă prima verificare accesează o locaţie ocupată; prin urmare,

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) \, .$$

În general, a i-a verificare este necesară doar dacă primele $i-1$ verificări accesează locaţii ocupate şi locaţia examinată este cu aceeaşi probabilitate oricare dintre cele $m-i+1$ locaţii rămase, din care $n-i+1$ sunt ocupate. Deci,

$$q_i = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) ... \left(\frac{n-i+1}{m-i+1}\right) \leq \left(\frac{n}{m}\right)^i = \alpha^i$$

pentru $i = 1, 2, ..., n$, pentru că $(n-j)/(m-j) \leq n/m$ dacă $n \leq m$ şi $j \geq 0$. După n verificări, toate cele n locaţii ocupate au fost vizitate şi nu vor fi verificate din nou şi astfel $q_i = 0$ pentru $i > n$.

Acum se poate evalua relaţia (12.6). Presupunând $\alpha < 1$, numărul mediu de verificări într-o căutare fără succes este

$$1 + \sum_{i=0}^{\infty} ip_i = 1 + \sum_{i=1}^{\infty} q_i \leq 1 + \alpha + \alpha^2 + \alpha^3 + ... = \frac{1}{1-\alpha} \, . \tag{12.7}$$

Relaţia (12.7) are o interpretare intuitivă: prima verificare se face întotdeauna, a doua verificare este necesară cu o probabilitate aproximativ egală cu α, a treia verificare este necesară cu o probabilitate aproximativ egală cu α^2 şi aşa mai departe. ∎

Dacă α este o constantă, teorema 12.5 prognozează faptul că o căutare fără succes necesită un timp $O(1)$. De exemplu, dacă tabela de dispersie este pe jumătate plină, numărul mediu de verificări într-o căutare fără succes este cel mult $1/(1-.5) = 2$. Dacă este 90% ocupată, numărul mediu de verificări este cel mult $1/(1-.9) = 10$.

Teorema 12.5 ne dă, aproape imediat, performanţa procedurii DISPERSIE-INSEREAZĂ.

Corolarul 12.6 Inserarea unui element într-o tabelă de dispersie cu adresare deschisă, cu factorul de încărcare α necesită , în cazul mediu, cel mult $1/(1-\alpha)$ verificări, în ipoteza dispersiei uniforme.

Demonstraţie. Un element este inserat doar dacă există spaţiu în tabelă şi, deci, $\alpha < 1$. Inserarea unei chei necesită o căutare fără succes, urmată de o plasare a cheii în prima locaţie găsită liberă. Prin urmare, numărul mediu de verificări este cel mult $1/(1-\alpha)$. ∎

Calculul numărului mediu de verificări pentru o căutare cu succes necesită un pic mai mult de lucru.

Teorema 12.7 Fiind dată o tabelă de dispersie cu adresare deschisă, cu factorul de încărcare $\alpha < 1$, numărul mediu de verificări într-o căutare cu succes este cel mult

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} ,$$

în ipoteza dispersiei uniforme şi presupunând că fiecare cheie din tabelă este căutată cu aceeaşi probabilitate.

Demonstraţie. O căutare pentru o cheie k urmează aceeaşi secvenţă de verificare ce a fost urmată când elementul având cheia k a fost inserat. Conform corolarului 12.6, dacă k a fost a $i+1$-a cheie inserată în tabela de dispersie, numărul mediu de verificări făcute într-o căutare a lui k este cel mult $1/(1-i/m) = m(m-i)$. Calculând media peste toate cele n chei din tabela de dispersie, obţinem numărul mediu de verificări dintr-o căutare cu succes:

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha}(H_m - H_{m-n}) ,$$

unde $H_i = \sum_{j=1}^{i} 1/j$ este al i-lea număr armonic (definit ca în relaţia (3.5)). Folosind marginile $\ln i \le H_i \le \ln i + 1$ din relaţiile (3.11) şi (3.12), obţinem

$$\begin{aligned} \frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha} \sum_{k=m-n+1}^{m} 1/k \le \frac{1}{\alpha} \int_{m-n}^{m} (1/x)dx \\ &= \frac{1}{\alpha} \ln(m/(m-n)) = \frac{1}{\alpha} \ln(1/(1-\alpha)) \end{aligned}$$

ca mărginire pentru numărul mediu de verificări într-o căutare cu succes. ∎

Dacă tabela de dispersie este pe jumătate ocupată, numărul mediu de verificări într-o căutare cu succes este mai mic decât 1,387. Dacă tabela de dispersie este 90% ocupată, numărul mediu de verificări este mai mic decât 2,559.

Exerciţii

12.4-1 Se consideră că se inserează cheile 10, 22, 31, 4, 15, 28, 17, 88, 59 într-o tabelă de dispersie de lungime $m = 11$ folosind adresarea deschisă cu funcţia primară de dispersie $h'(k) = k \bmod m$. Ilustraţi rezultatul inserării acestor chei folosind verificarea liniară, verificarea pătratică cu $c_1 = 1$ şi $c_2 = 3$ şi folosind dispersia dublă cu $h_2(k) = 1 + (k \bmod (m-1))$.

12.4-2 Scrieţi un algoritm în pseudocod pentru DISPERSIE-ŞTERGE după descrierea din text şi modificaţi DISPERSIE-INSEREAZĂ şi DISPERSIE-CAUTĂ pentru a încorpora valoarea specială ŞTERS.

12.4-3 ⋆ Să presupunem că folosim dispersia dublă pentru a rezolva coliziunile; mai exact, folosim funcţia de dispersie $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$. Arătaţi că secvenţa de verificare $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$ este o permutare a secvenţei de locaţii $\langle 0, 1, ..., m-1\rangle$ dacă şi numai dacă $h_2(k)$ este relativ prim cu m. (*Indicaţie:* Vezi capitolul 33.)

12.4-4 Se consideră o tabelă de dispersie cu adresare deschisă şi cu dispersie uniformă şi factorul de încărcare $\alpha = 1/2$. Daţi margini superioare pentru numărul mediu de verificări într-o căutare fără succes şi pentru numărul mediu de verificări într-o căutare cu succes. Care este numărul mediu de verificări într-o căutare cu succes? Repetaţi calculele pentru factorii de încărcare $3/4$ şi $7/8$.

12.4-5 ⋆ Să presupunem că inserăm n chei într-o tabelă de dispersie de dimensiune m folosind adresarea deschisă şi dispersia uniformă. Notăm $cup(n,m)$ probabilitatea de a nu apare nici o coliziune. Arătaţi că $p(n,m) \leq e^{-n(n-1)/2m}$. (*Indicaţie:* Vezi relaţia (2.7).) Argumentaţi că atunci când n depăşeşte $\sqrt{m}$, probabilitatea evitării coliziunilor descreşte rapid către zero.

12.4-6 ⋆ Se consideră o tabelă de dispersie cu adresare deschisă având factorul de încărcare α. Determinaţi acea valoare nenulă a lui α pentru care numărul mediu de verificări într-o căutare fără succes este de două ori mai mare decât numărul mediu de verificări dintr-o căutare cu succes. Folosiţi estimarea $(1/\alpha)ln(1/(1-\alpha))$ pentru numărul de verificări necesare într-o căutare cu succes.

Probleme

12-1 Marginea celei mai lungi verificări pentru dispersie
O tabelă de dispersie de dimensiune m este folosită pentru a memora n obiecte, cu $n \leq m/2$. Pentru rezolvarea coliziunilor se foloseşte adresarea deschisă.

- ***a.*** În ipoteza dispersiei uniforme, arătaţi că pentru $i = 1, 2, ..., n$, probabilitatea ca a i-a inserare să necesite strict mai mult de k verificări este cel mult 2^{-k}.
- ***b.*** Arătaţi că pentru $i = 1, 2, ..., n$, probabilitatea ca a i-a inserare să necesite mai mult de $2 \lg n$ verificări este cel mult $1/n^2$.

Fie variabila aleatoare X_i, semnificând numărul de verificări necesare pentru a i-a inserare. Aţi arătat la punctul (b) că $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Fie variabila aleatoare $X = \max_{1 \leq i \leq n} X_i$ care semnifică numărul maxim de verificări necesare oricăreia din cele n inserări.

c. Arătaţi că $\Pr\{X > 2\lg n\} \leq 1/n$.

d. Arătaţi că lungimea medie a celei mai lungi secvenţe de verificare este $\mathrm{E}[X] = O(\lg n)$.

12-2 Căutarea într-o mulţime statică

Se cere să se implementeze o mulţime dinamică având n elemente, în care cheile sunt numere. Mulţimea este statică (nu există operaţiile INSEREAZĂ şi ŞTERGE) şi singura operaţie necesară este CAUTĂ. Se pune la dispoziţie un timp arbitrar pentru a preprocesa cele n elemente, astfel încât operaţiile de căutare să se execute rapid.

a. Arătaţi că procedura CAUTĂ poate fi implementată astfel încât, în cazul cel mai defavorabil să se execute într-un timp $O(\lg n)$, fără a folosi spaţiu suplimentar faţă de cel necesar memorării elementelor.

b. Se consideră implementarea mulţimii printr-o tabelă de dispersie cu adresare deschisă având m locaţii care foloseşte dispersia uniformă. Care este spaţiul minim necesar (de lungime $m-n$) suplimentar pentru ca performanţa, în cazul mediu a unei căutări fără succes, să fie cel puţin la fel de bună ca marginea de la punctul (a)? Răspunsul trebuie să fie o mărginire asimptotică a lui $m - n$ în funcţie de n.

12-3 Margini pentru mărimea locaţiei la înlănţuire

Să presupunem că avem o tabelă de dispersie cu n locaţii, având coliziuni rezolvate prin înlănţuire şi să presupunem că se inserează n chei în tabelă. Fiecare cheie se poate dispersa, cu aceeaşi probabilitate, în oricare dintre locaţii. Fie M numărul maxim de chei într-o locaţie, după ce toate cheile au fost inserate. Demonstraţi că $\mathrm{E}[M]$, valoarea medie a lui M, are o margine superioară egală cu $O(\lg n/\lg\lg n)$.

a. Fie Q_k probabilitatea ca un număr de k chei să se disperseze într-o locaţie particulară. Argumentaţi că probabilitatea Q_k este dată de

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

b. Fie P_k probabilitatea ca $M = k$, adică probabilitatea ca locaţia care conţine cele mai multe chei să conţină k chei. Arătaţi că $P_k \leq nQ_k$.

c. Folosiţi aproximarea Stirling, din relaţia (2.11), pentru a arăta că $Q_k < e^k/k^k$.

d. Arătaţi că există o constantă $c > 1$ astfel încât $Q_{k_0} < 1/n^3$ pentru $k_0 = c\lg n/\lg\lg n$. Deduceţi că $P_k < 1/n^2$ pentru $k = c\lg n/\lg\lg n$.

e. Arătaţi că

$$\mathrm{E}[M] \leq \Pr\left\{M > \frac{c\lg n}{\lg\lg n}\right\} \cdot n + \Pr\left\{M \leq \frac{c\lg n}{\lg\lg n}\right\} \cdot \frac{c\lg n}{\lg\lg n}.$$

Deduceţi că $E[M] = O(\lg n/\lg\lg n)$.

12-4 Verificarea pătratică
Să presupunem că avem o cheie k, care trebuie căutată într-o tabelă de dispersie având poziţiile $0, 1, ..., m-1$ şi să presupunem că avem o funcţie de dispersie h care transformă spaţiul cheilor în mulţimea $\{0, 1, ..., m-1\}$. Schema de căutare este următoarea.

1. Se calculează valoarea $i \leftarrow h(k)$ şi se iniţializează $j \leftarrow 0$.
2. Se verifică poziţia i pentru cheia dorită k. Dacă se găseşte sau dacă poziţia este liberă atunci căutarea se termină.
3. Se fac atribuirile $j \leftarrow (i + j) \bmod m$ şi se revine la pasul 2.

Se presupune că m este o putere a lui 2.

a. Arătaţi că această schemă este o instanţiere a schemei generale de "verificare pătratică" evidenţiind valorile specifice ale constantelor c_1 şi c_2 din relaţia (12.5).

b. Demonstraţi că acest algoritm examinează fiecare poziţie din tabelă în cazul cel mai defavorabil.

12-5 dispersie k-universală
Fie $\mathcal{H} = \{h\}$ o clasă de funcţii de dispersie în care fiecare h transformă universul U al cheilor în $\{0, 1, ..., m-1\}$. Spunem că $\mathcal{H}$ este ***k-universală*** dacă pentru fiecare secvenţă fixă de k chei distincte $\langle x_1, x_2, ..., x_k \rangle$ şi pentru fiecare funcţie h aleasă aleator din $\mathcal{H}$, secvenţa $\langle h(x_1), h(x_2), ..., h(x_k) \rangle$ este, cu aceeaşi probabilitate, oricare dintre cele m^k secvenţe de lungime k având elemente din $\{0, 1, ..., m-1\}$.

a. Arătaţi că dacă $\mathcal{H}$ este 2-universală atunci ea este universală.

b. Arătaţi că clasa $\mathcal{H}$ definită în secţiunea 12.3.3 nu este 2-universală.

c. Arătaţi că dacă modificăm definiţia lui $\mathcal{H}$ din secţiunea 12.3.3 astfel încât fiecare funcţie să conţină un termen constant b, adică dacă înlocuim $h(x)$ cu

$$h_{a,b}(x) = \sum_{i=0}^{r} a_i x_i + b \bmod m \,,$$

atunci $\mathcal{H}$ este 2-universală.

Note bibliografice

Knuth [123] şi Gonnet [90] sunt referinţe excelente pentru algoritmi de dispersie. Knuth îl creditează pe H. P. Luhn (1953) cu inventarea tabelelor de dispersie, împreună cu metoda înlănţuirii pentru rezolvarea coliziunilor. Aproximativ în aceeaşi perioadă, G. M. Amdahl a avut ideea originală a adresării deschise.

13 Arbori binari de căutare

Arborii de căutare sunt structuri de date ce posedă multe operaţii specifice mulţimilor dinamice, între care menţionăm CAUTĂ, MINIM, MAXIM, PREDECESOR, SUCCESOR, INSEREAZĂ şi ŞTERGE. Prin urmare un arbore de căutare se poate folosi atât ca dicţionar, cât şi pe post de coadă cu priorităţi.

Operaţiile de bază pe arborii binari consumă un timp proporţional cu înălţimea arborelui. Pentru un arbore binar complet cu n noduri, aceste operaţii se execută în cazul cel mai defavorabil într-un timp $\Theta(\lg n)$. Dacă însă arborele este un lanţ liniar de n noduri, atunci timpul consumat în cazul cel mai defavorabil este $\Theta(n)$. În secţiunea 13.4 vom vedea că înălţimea unui arbore binar de căutare construit aleator este $O(\lg n)$, deci operaţiile de bază pe mulţimile dinamice vor consuma un timp de $\Theta(\lg n)$.

În practică nu se poate garanta că arborii binari de căutare se construiesc aleator, însă există diverse tipuri speciale de astfel de structuri de date pentru care se garantează performanţe bune ale operaţiilor de bază chiar şi în cazurile cele mai defavorabile. Astfel, în capitolul 14 se prezintă un astfel de caz special, arborii roşu-negru, care au înălţimea $O(\lg n)$. Capitolul 19 introduce B-arborii, care se folosesc în special la întreţinerea bazelor de date cu acces direct, memorate pe suport magnetic extern.

După prezentarea proprietăţilor de bază ale arborilor binari de căutare, secţiunile următoare vor ilustra cum se traversează un astfel de arbore în scopul listării valorilor sale în ordine crescătoare, cum se caută o valoare memorată în arborele binar de căutare, cum se găseşte elementul minim sau maxim, cum se determină succesorul sau predecesorul unui element şi cum se inserează (adaugă) sau se şterge un element. Proprietăţile matematice de bază ale arborilor au fost prezentate în capitolul 5.

13.1. Ce este un arbore binar de căutare?

Aşa cum sugerează numele său, un arbore binar de căutare este organizat sub formă de arbore binar, aşa cum se observă în figura 13.1. Un astfel de arbore se poate reprezenta printr-o structură de date înlănţuită, în care fiecare nod este un obiect. Pe lângă un câmp *cheie* şi date adiţionale, fiecare obiect nod conţine câmpurile *stânga*, *dreapta* şi p care punctează spre (referă) nodurile corespunzătoare fiului stâng, fiului drept şi respectiv părintelui nodului. Dacă un fiu sau un părinte lipseşte, câmpul corespunzător acestuia va conţine valoarea NIL. Nodul rădăcină este singurul nod din arbore care are valoarea NIL pentru câmpul părinte p.

Într-un arbore binar de căutare, cheile sunt întotdeauna astfel memorate încât ele satisfac ***proprietatea arborelui binar de căutare***:

> Fie x un nod dintr-un arbore binar de căutare. Dacă y este un nod din subarborele stâng al lui x, atunci $cheie[y] \leq cheie[x]$. Dacă y este un nod din subarborele drept al lui x, atunci $cheie[x] \leq cheie[y]$.

Astfel, în figura 13.1(a) cheia rădăcinii este 5, iar cheile 2, 3 şi 5 din subarborele stâng nu sunt mai mari decât 5, pe când cheile 7 şi 8 din subarborele drept nu sunt mai mici decât 5. Aceeaşi

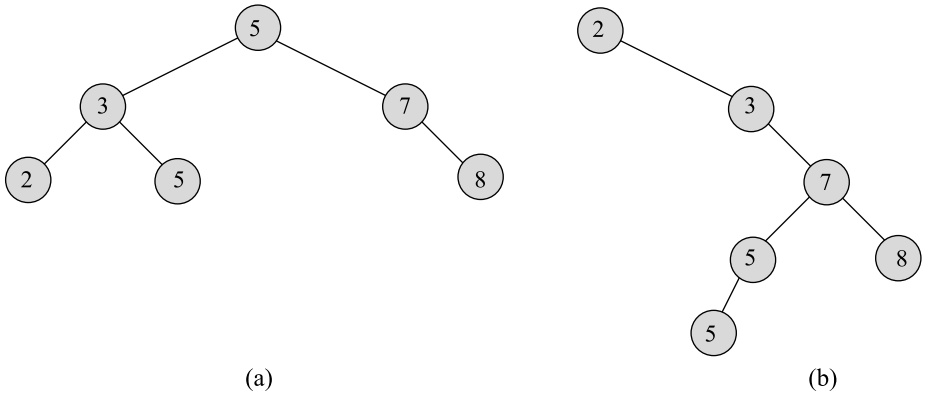


Figura 13.1 Arbori binari de căutare. Pentru orice nod x, cheile din subarborele stâng al lui x au valoarea mai mică sau egală cu $cheie[x]$, iar cheile din subarborele drept al lui x au valoarea mai mare sau egală cu $cheie[x]$. Aceeaşi mulţime de valori se poate reprezenta prin arbori binari de căutare, diferiţi. Timpul de execuţie, în cazul cel mai defavorabil, pentru majoritatea operaţiilor arborilor de căutare este proporţional cu înălţimea arborelui. (**a**) Un arbore binar de căutare cu 6 noduri şi de înălţime 2. (**b**) Un arbore binar de căutare mai puţin eficient care conţine aceleaşi chei şi are înălţimea 4.

proprietate se verifică pentru fiecare nod din arbore. De exemplu, cheia 3 din figura 13.1(a) nu este mai mică decât cheia 2 din subarborele său stâng şi nu este mai mare decât cheia 5 din subarborele său drept.

Proprietatea arborelui binar de căutare ne permite să tipărim toate cheile în ordine crescătoare cu ajutorul unui algoritm recursiv simplu, numit ***traversarea arborelui în inordine***. Numele acestui algoritm derivă din faptul că cheia rădăcinii unui subarbore se tipăreşte între valorile din subarborele său stâng şi cele din subarborele său drept. (Similar, o ***traversare a arborelui în preordine*** va tipări cheia rădăcinii înaintea cheilor din subarbori, iar o ***traversare a arborelui în postordine*** va tipări cheia rădăcinii după cheile din subarbori). Pentru a folosi procedura următoare în scopul tipăririi tuturor elementelor din arborele binar de căutare T o vom apela cu ARBORE-TRAVERSARE-INORDINE(*rădăcină*$[T]$).

ARBORE-TRAVERSARE-INORDINE(x)
1: **dacă** $x \neq$ NIL **atunci**
2: ARBORE-TRAVERSARE-INORDINE(*stânga*$[x]$)
3: afişează $cheie[x]$
4: ARBORE-TRAVERSARE-INORDINE($dreapta[x]$)

Spre exemplu, traversarea în inordine a arborelui afişează cheile fiecăruia dintre arborii binari de căutare din figura 13.1 în ordinea 2, 3, 5, 5, 7, 8. Corectitudinea algoritmului se demonstrează prin inducţie folosind direct proprietatea arborelui binar de căutare. Deoarece după apelul iniţial procedura se apelează recursiv de exact două ori pentru fiecare nod din arbore – o dată pentru fiul său stâng şi încă o dată pentru fiul său drept – rezultă că este nevoie de un timp $\Theta(n)$ pentru a traversa un arbore binar de căutare cu n noduri.

Exerciţii

13.1-1 Desenaţi arbori binari de căutare de înălţime 2, 3, 4, 5 şi respectiv 6 pentru mulţimea de chei {1, 4, 5, 10, 16, 17, 21}.

13.1-2 Care este deosebirea dintre proprietatea arborelui binar de căutare şi proprietatea de ansamblu (heap) (7.1)? Se poate folosi proprietatea de ansamblu pentru a afişa cheile unui arbore având n noduri în ordine crescătoare într-un timp $O(n)$? Explicaţi cum sau de ce nu.

13.1-3 Daţi un algoritm nerecursiv care efectuează o traversare în inordine a unui arbore. (*Indicaţie:* Există o soluţie simplă care foloseşte o stivă pe post de structură de date auxiliară şi o soluţie mai complicată dar în acelaşi timp mai elegantă care nu mai foloseşte stiva, însă presupune că există o operaţie care realizează testul de egalitate a doi pointeri.)

13.1-4 Daţi algoritmi recursivi care realizează traversarea în preordine şi postordine a unui arbore cu n noduri într-un timp $\Theta(n)$.

13.1-5 Argumentaţi că, deoarece sortarea a n elemente, care foloseşte modelul comparaţiei are nevoie de un timp de $\Omega(n \lg n)$, în cazul cel mai defavorabil, orice algoritm pentru construirea unui arbore binar de căutare care se bazează pe comparaţii şi care foloseşte la intrare o listă arbitrară având n elemente, necesită tot un timp de $\Omega(n \lg n)$ în cazul cel mai defavorabil.

13.2. Interogarea într-un arbore binar de căutare

Principala operaţie efectuată pe un arbore binar de căutare este căutarea unei chei memorate în acesta. În afară de operaţia CAUTĂ, arborii binari de căutare permit şi alte interogări precum MINIM, MAXIM, SUCCESOR şi PREDECESOR. În această secţiune vom examina aceste operaţii şi vom arăta că fiecare dintre ele se poate executa într-un timp $O(h)$ pe un arbore binar de căutare de înălţime h.

Căutarea

Procedura următoare poate fi folosită la căutarea unui nod având cheia cunoscută, într-un arbore binar de căutare. Fiind date un pointer x la rădăcina arborelui şi o valoare k a cheii, ARBORE-CAUTĂ returnează un pointer la nodul având cheia k dacă există un asemenea nod în arbore sau NIL în caz contrar.

ARBORE-CAUTĂ(x, k)
1: **dacă** x = NIL sau $k = cheie[x]$ **atunci**
2: **returnează** x
3: **dacă** $k < cheie[x]$ **atunci**
4: **returnează** ARBORE-CAUTĂ(*stânga*$[x], k)$
5: **altfel**
6: **returnează** ARBORE-CAUTĂ$(dreapta[x], k)$

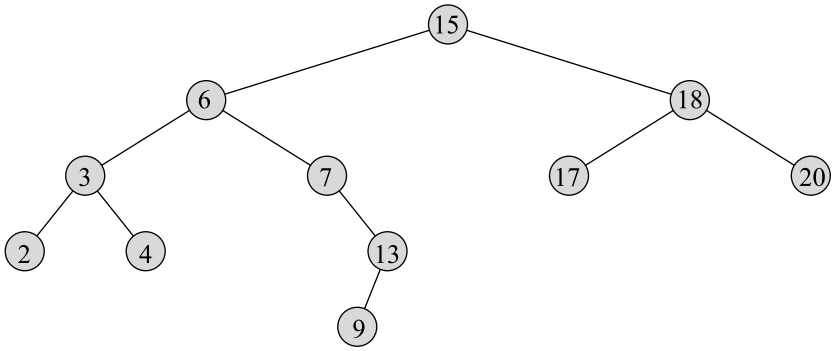


Figura 13.2 Interogări într-un arbore binar de căutare. Pentru a căuta în arbore cheia 13, se parcurge, începând cu rădăcina, drumul $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$. Cheia minimă din arbore este 2, care se determină pornind de la rădăcină şi urmând pointerii *stânga*. Cheia maximă din arbore este 20, determinată începând cu rădăcina şi urmând pointerii *dreapta*. Succesorul nodului având cheia 15 este nodul având cheia 17, deoarece această cheie este cheia minimă din subarborele drept al nodului având cheia 15. Nodul având cheia 13 nu are subarbore drept, prin urmare succesorul său este cel mai de jos nod strămoş al său al cărui fiu stâng este de asemenea strămoş pentru 13. În cazul nodului având cheia 13, succesorul este nodul având cheia 15.

Procedura începe căutarea cu rădăcina şi trasează un drum în jos în arbore, aşa cum se ilustrează în figura 13.2. Pentru fiecare nod x întâlnit, se compară cheia k cu $cheie[x]$. Dacă cele două chei sunt egale, căutarea s-a terminat. Dacă k are o valoare mai mică decât $cheie[x]$, căutarea continuă în subarborele stâng al lui x, deoarece din proprietatea arborelui binar de căutare rezultă că k nu poate fi memorat în subarborele drept al lui x. Simetric, dacă k este mai mare decât $cheie[x]$, căutarea continuă în subarborele drept al lui x. Nodurile traversate în timpul aplicării recursive a acestei proceduri formează un drum în jos, pornind de la rădăcină, deci timpul de execuţie al procedurii ARBORE-CAUTĂ este $O(h)$, unde h este înălţimea arborelui.

Aceeaşi procedură se poate scrie iterativ, prin "derularea" recursivităţii într-un ciclu **cât timp**. Versiunea iterativă este mai eficientă pe majoritatea calculatoarelor.

ARBORE-CAUTĂ-ITERATIV(x, k)
1: **cât timp** x $\neq$ NIL şi $k \neq cheie[x]$ **execută**
2: **dacă** $k < cheie[x]$ **atunci**
3: $x \leftarrow$ *stânga* $[x]$
4: **altfel**
5: $x \leftarrow dreapta[x]$
6: **returnează** x

Minimul şi maximul

Determinarea elementului având cheia minimă dintr-un arbore binar de căutare se realizează întotdeauna urmând pointerii fiu *stânga* începând cu rădăcina şi terminând când se întâlneşte NIL, aşa cum se ilustrează în figura 13.2. Procedura următoare întoarce un pointer la elementul minim din subarborele a cărui rădăcină este nodul x.

ARBORE-MINIM(x)
1: **cât timp** *stânga*$[x] \neq$ NIL **execută**
2: $x \leftarrow$ *stânga*$[x]$
3: **returnează** x

Proprietatea arborelui binar de căutare garantează corectitudinea algoritmului ARBORE-MINIM. Dacă un nod x nu are subarbore stâng, atunci din faptul că fiecare cheie din subarborele său drept este cel puţin la fel de mare ca şi $cheie[x]$ rezultă că cheia minimă a subarborelui având rădăcina x este chiar $cheie[x]$. Dacă nodul x are un subarbore stâng, atunci deoarece orice cheie din subarborele său drept este mai mare sau egală cu $cheie[x]$ şi orice cheie din subarborele său stâng este mai mică sau egală cu $cheie[x]$, rezultă că cheia minimă din subarborele având rădăcina x este cheia minimă a subarborelui având rădăcina *stânga*$[x]$.

Pseudocodul pentru procedura ARBORE-MAXIM este simetric.

ARBORE-MAXIM(x)
1: **cât timp** $dreapta[x] \neq$ NIL **execută**
2: $x \leftarrow dreapta[x]$
3: **returnează** x

Deoarece ambele proceduri trasează drumuri în jos în arbore, ele se execută într-un timp $O(h)$, unde h este înălţimea arborelui în care se face căutarea.

Succesorul şi predecesorul unui nod

Cunoscând un nod dintr-un arbore binar de căutare, este important câteodată să putem determina succesorul său în ordinea de sortare determinată de traversarea în inordine a arborelui. Dacă toate cheile sunt distincte, succesorul unui nod x este nodul având cea mai mică cheie mai mare decât $cheie[x]$. Structura de arbore binar de căutare permite determinarea succesorului unui nod chiar şi fără compararea cheilor. Procedura următoare returnează succesorul unui nod x dintr-un arbore binar de căutare, dacă succesorul există, sau NIL, dacă x are cea mai mare cheie din arbore.

ARBORE-SUCCESOR(x)
1: **dacă** $dreapta[x] \neq$ NIL **atunci**
2: **returnează** ARBORE-MINIM($dreapta[x]$)
3: $y \leftarrow p[x]$
4: **cât timp** $y \neq$ NIL şi $x = dreapta[y]$ **execută**
5: $x \leftarrow y$
6: $y \leftarrow p[y]$
7: **returnează** y

Codul procedurii ARBORE-SUCCESOR tratează două alternative. Dacă subarborele drept al nodului x nu este vid, atunci succesorul lui x este chiar cel mai din stânga nod din acest subarbore drept, care este determinat în linia 2 prin apelul ARBORE-MINIM($dreapta[x]$). De exemplu, succesorul nodului cu cheia 15 din figura 13.2 este nodul cu cheia 17.

În celălalt caz, când subarborele drept al nodului x este vid şi x are un succesor y, atunci y este cel mai de jos strămoş al lui x al cărui fiu din stânga este de asemenea strămoş al lui x.

În figura 13.2, succesorul nodului cu cheia 13 este nodul cu cheia 15. Pentru a-l determina pe y, se traversează arborele de la x în sus până când se întâlneşte un nod care este fiul stâng al părintelui său; acest lucru se realizează în liniile 3–7 ale codului procedurii ARBORE-SUCCESOR.

În cazul unui arbore de înălţime h, timpul de execuţie al procedurii ARBORE-SUCCESOR este $O(h)$ deoarece în ambele situaţii se urmează fie un drum în josul arborelui, fie unul în susul arborelui. Procedura ARBORE-PREDECESOR, care este similară, se execută tot într-un timp $O(h)$.

Rezultatele obţinute până acum se pot rezuma în teorema următoare.

Teorema 13.1 Pe un arbore binar de căutare de înălţime h, operaţiile pe mulţimi dinamice CAUTĂ, MINIM, MAXIM, SUCCESOR şi PREDECESOR se pot executa într-un timp $O(h)$.

■

Exerciţii

13.2-1 Presupunem că avem memorate într-un arbore binar de căutare numere cuprinse între 1 şi 1000 şi că dorim să căutăm numărul 363. Care dintre următoarele şiruri *nu* poate fi secvenţa nodurilor examinate?

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

13.2-2 Profesorul Bunyan crede că a descoperit o proprietate remarcabilă a arborilor binari de căutare. Să presupunem că determinarea cheii k într-un astfel de arbore se termină într-o frunză. Împărţind cheile din nodurile arborelui în trei mulţimi A, B şi C construite astfel: A conţine cheile nodurilor de la stânga drumului de căutare, B – cheile de pe drumul de căutare, iar C – cheile de la dreapta drumului de căutare, profesorul Bunyan afirmă că oricare trei chei $a \in A$, $b \in B$ şi $c \in C$ trebuie să îndeplinească condiţia $a \leq b \leq c$. Daţi un contraexemplu cât mai mic, care să infirme proprietatea descoperită de profesor.

13.2-3 Folosiţi proprietatea arborelui binar de căutare pentru a demonstra riguros corectitudinea codului pentru procedura ARBORE-SUCCESOR.

13.2-4 Traversarea în inordine pentru un arbore binar de căutare având n noduri se poate implementa prin găsirea elementului minim din arbore cu procedura ARBORE-MINIM şi apoi apelând de $n - 1$ ori procedura ARBORE-SUCCESOR. Demonstraţi că acest algoritm se execută într-un timp de $\Theta(n)$.

13.2-5 Demonstraţi că, într-un arbore binar de căutare de înălţime h, prin k apeluri succesive ale procedurii ARBORE-SUCCESOR, se consumă un timp $O(k + h)$, indiferent care este nodul de pornire.

13.2-6 Fie T un arbore binar de căutare având toate cheile distincte, x un nod frunză şi y părintele său. Arătaţi că $cheie[y]$ este fie cea mai mică cheie din T care este mai mare decât $cheie[x]$ fie cea mai mare cheie din T care este mai mică decât $cheie[x]$.

13.3. Inserarea şi ştergerea

Operaţiile de inserare şi ştergere provoacă modificarea mulţimii dinamice reprezentată de arborele binar de căutare. Structura de date trebuie modificată în sensul că ea trebuie pe de o parte să reflecte inserarea sau ştergerea, iar pe de altă parte să conserve proprietatea arborelui binar de căutare. Aşa cum vom vedea în cele ce urmează, modificarea unui arbore pentru a insera un element nou, este relativ simplă (directă), pe când gestiunea ştergerii unui element este mai complicată.

Inserarea

Vom folosi procedura ARBORE-INSEREAZĂ pentru a insera o nouă valoare v într-un arbore binar de căutare T. Procedurii i se transmite un nod z pentru care $cheie[z] = v$, $stânga[z] = \text{NIL}$ şi $dreapta[z] = \text{NIL}$. Ea va modifica arborele T şi unele dintre câmpurile lui z astfel încât z va fi inserat pe poziţia corespunzătoare în arbore.

```
ARBORE-INSEREAZĂ(T, z)
 1: y ← NIL
 2: x ←rădăcină[T]
 3: cât timp x ≠ NIL execută
 4:    y ← x
 5:    dacă cheie[z] < cheie[x] atunci
 6:       x ←stânga[x]
 7:    altfel
 8:       x ← dreapta[x]
 9: p[z] ← y
10: dacă y = NIL atunci
11:    rădăcină[T] ← z
12: altfel dacă cheie[z] < cheie[y] atunci
13:    stânga[y] ← z
14: altfel
15:    dreapta[y] ← z
```

Figura 13.3 ilustrează modul de lucru al procedurii ARBORE-INSEREAZĂ. Ca şi ARBORE-CAUTĂ şi ARBORE-CAUTĂ-ITERATIV, ARBORE-INSEREAZĂ începe cu rădăcina arborelui şi trasează un drum în jos. Pointerul x va referi nodul din drum, iar pointerul y va referi părintele lui x. După iniţializare, ciclul **cât timp** din liniile 3–8 asigură deplasarea acestor pointeri în josul arborelui, selectarea alternativei la stânga sau la dreapta realizându-se în funcţie de rezultatul comparării cheilor $cheie[z]$ şi $cheie[x]$, până când x este setat pe NIL. Acest NIL ocupă poziţia în care trebuie să fie plasat elementul z. Liniile 9–15 realizează setarea pointerilor pentru a realiza efectiv inserarea lui z.

Ca şi celelalte operaţii primitive pe arborii de căutare, procedura ARBORE-INSEREAZĂ se execută într-un timp $O(h)$ pe un arbore de înălţime h.

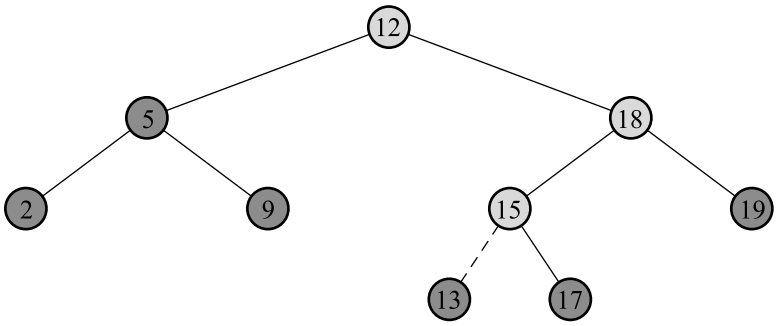


Figura 13.3 Inserarea unui element având cheia 13 într-un arbore binar de căutare. Nodurile haşurate deschis indică drumul în jos, de la rădăcină la poziţia în care se inserează elementul. Linia întreruptă semnifică legătura adăugată la arbore pentru a insera elementul.

Ştergerea

Procedura pentru ştergerea unui nod dat z dintr-un arbore binar de căutare primeşte ca argument un pointer la z. Procedura ia în considerare toate cele trei situaţii ilustrate în figura 13.4. Dacă z nu are fii, se va modifica părintele său $p[z]$ pentru a-i înlocui fiul z cu NIL. Dacă nodul are un singur fiu, z va fi eliminat din arbore prin inserarea unei legături de la părintele lui z la fiul lui z. În sfârşit, dacă nodul are doi fii, se va elimina din arbore succesorul y al lui z, care nu are fiu stâng (vezi exerciţiul 13.3-4) şi apoi se vor înlocui cheia şi datele adiţionale ale lui z cu cheia şi datele adiţionale ale lui y.

Pseudocodul procedurii ARBORE-ŞTERGE organizează puţin diferit aceste trei situaţii.

ARBORE-ŞTERGE(T, z)
1: **dacă** *stânga*$[z]$ = NIL sau $dreapta[z]$ = NIL **atunci**
2: $y \leftarrow z$
3: **altfel**
4: $y \leftarrow$ ARBORE-SUCCESOR(z)
5: **dacă** *stânga*$[y] \neq$ NIL **atunci**
6: $x \leftarrow$*stânga*$[y]$
7: **altfel**
8: $x \leftarrow dreapta[y]$
9: **dacă** $x \neq$ NIL **atunci**
10: $p[x] \leftarrow p[y]$
11: **dacă** $p[y]$ = NIL **atunci**
12: *rădăcină*$[T] \leftarrow x$
13: **altfel dacă** $y =$*stânga*$[p[y]]$ **atunci**
14: *stânga*$[p[y]] \leftarrow x$
15: **altfel**
16: $dreapta[p[y]] \leftarrow x$
17: **dacă** $y \neq z$ **atunci**
18: $cheie[z] \leftarrow cheie[y]$ ▷ se copiază şi datele adiţionale ale lui y
19: **returnează** y

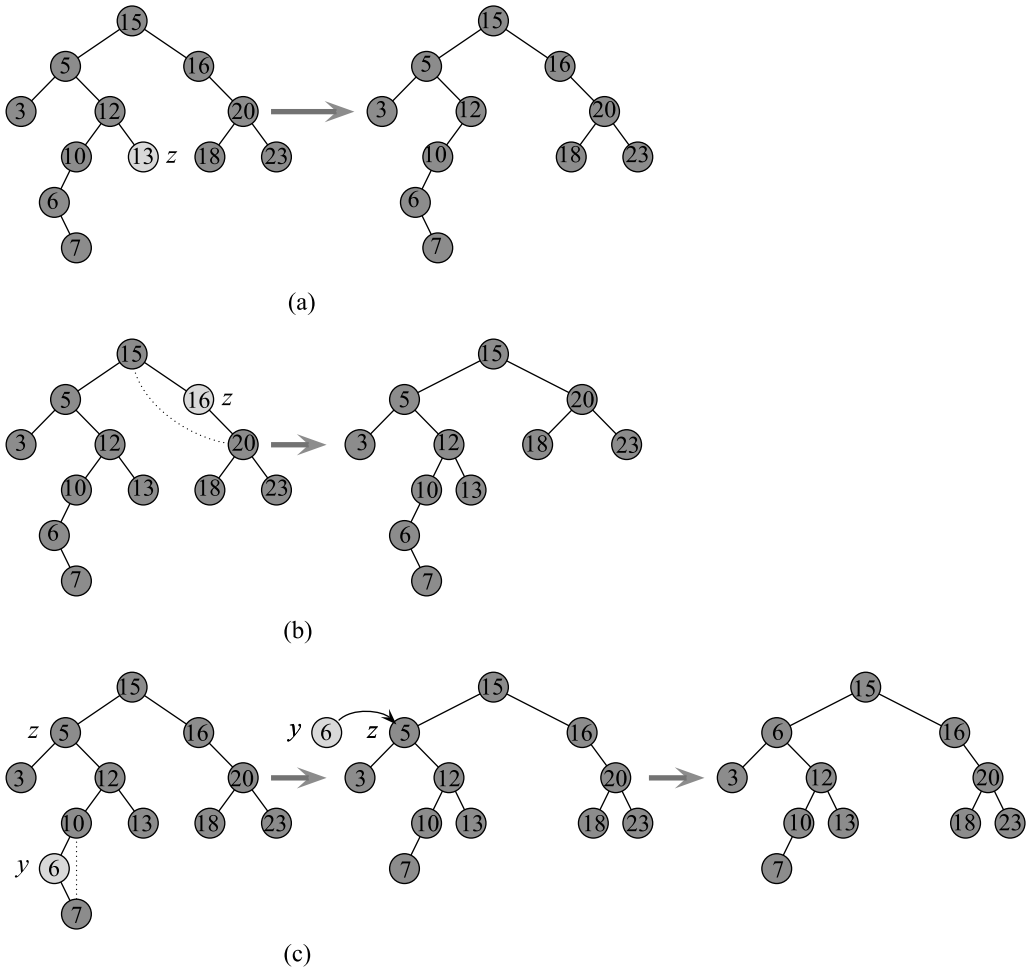


Figura 13.4 Ştergerea unui nod z dintr-un arbore binar de căutare. În fiecare situaţie, nodul care se şterge este haşurat mai deschis decât celelalte. (**a**) Dacă z nu are fii, pur şi simplu se şterge. (**b**) Dacă z are un singur fiu, z se elimină din arbore. (**c**) Dacă z are doi fii, se elimină succesorul y al său, care are cel mult un fiu, şi apoi se înlocuieşte cheia şi datele adiţionale ale lui z cu cheia şi datele adiţionale ale lui y.

În liniile 1–4, algoritmul determină nodul y care se va scoate din arbore. Nodul y este fie chiar nodul de intrare z (dacă z are cel mult un fiu) fie succesorul lui z (dacă z are doi fii). Apoi, în liniile 5–8, x se setează fie la fiul diferit de NIL al lui y fie la NIL dacă y nu are fii. Nodul y este eliminat din arbore în liniile 9–16 prin modificarea pointerilor din $p[y]$ şi x. Eliminarea lui y este oarecum complicată de necesitatea unei gestionări adecvate a cazurilor extreme când $x =$ NIL sau când y este rădăcina arborelui. În sfârşit, liniile 17–18 se vor executa când a fost eliminat succesorul y al lui z şi ele realizează mutarea cheii şi datelor adiţionale ale lui y în z, scriind peste vechiul conţinut al cheii şi al datelor adiţionale. Nodul y este întors ca rezultat de

către procedură în linia 19, pentru ca procedura apelantă să poată să-l recicleze, punându-l în lista de noduri libere. Procedura se execută într-un timp $O(h)$ pe un arbore de înălţime h.

Pe scurt, am demonstrat următoarea teoremă.

Teorema 13.2 Operaţiile pe mulţimi dinamice INSEREAZĂ şi ŞTERGE se pot executa într-un timp $O(h)$ pe un arbore de înălţime h.

■

Exerciţii

13.3-1 Daţi o versiune recursivă a procedurii ARBORE-INSEREAZĂ.

13.3-2 Presupunem că se construieşte un arbore binar de căutare prin inserarea repetată a unor valori distincte. Arătaţi că numărul de noduri examinate la căutarea unei valori în arbore este cu 1 mai mare decât numărul de noduri examinate la prima inserare a respectivei valori în arbore.

13.3-3 Fiind dată o mulţime de n numere, aceasta se poate sorta prin construirea unui arbore binar de căutare (folosind repetat ARBORE-INSEREAZĂ pentru inserarea numerelor unul câte unul) urmată de tipărirea numerelor prin traversarea în inordine a arborelui. Care sunt timpii de execuţie corespunzători celui mai defavorabil caz, respectiv celui mai favorabil caz pentru acest algoritm de sortare?

13.3-4 Arătaţi că dacă un nod dintr-un arbore binar de căutare are doi fii, atunci succesorul său nu are fiu stâng iar predecesorul său nu are fiu drept.

13.3-5 Fie o altă structură de date care conţine un pointer la un nod y dintr-un arbore binar de căutare. Ce probleme pot să apară dacă predecesorul z al lui y este şters din arbore cu procedura ARBORE-ŞTERGE? Cum s-ar putea rescrie această procedură pentru a elimina dificultăţile apărute?

13.3-6 Operaţia de ştergere este comutativă (în sensul că dacă se şterge prima dată nodul x şi apoi nodul y se obţine acelaşi rezultat ca şi atunci când se şterge prima dată nodul y şi apoi nodul x)? Demonstraţi sau daţi un contraexemplu.

13.3-7 Când nodul z din ARBORE-ŞTERGE are doi fii, se poate elimina din arbore predecesorul său şi nu succesorul. Unii au demonstrat că o strategie echitabilă, care dă aceeaşi prioritate (atenţie) predecesorului şi succesorului produce performanţe empirice mai bune. Cum ar trebui modificat algoritmul ARBORE-ŞTERGE pentru a implementa o astfel de strategie echitabilă?

13.4. Arbori binari de căutare construiţi aleator

Am demonstrat că toate operaţiile de bază pe un arbore binar de căutare se execută într-un timp $O(h)$, h fiind înălţimea arborelui. Înălţimea unui arbore binar de căutare se schimbă continuu când se fac inserări şi ştergeri de noduri în/din arbore. Pentru a analiza comportamentul arborilor binari de căutare, în practică, va trebui să facem anumite ipoteze statistice asupra distribuţiei cheilor şi asupra şirului de inserări şi ştergeri.

Din păcate, se ştie puţin despre înălţimea medie a unui arbore binar de căutare în cazul când se folosesc atât inserări cât şi ştergeri pentru a-l crea. Dacă arborele ar fi creat numai prin inserări, analiza ar fi mai simplă. Vom defini prin urmare noţiunea de ***arbore binar de căutare construit aleator*** cu n chei distincte, ca fiind acel arbore care se construieşte prin inserarea într-o ordine aleatoare a cheilor într-un arbore iniţial vid, considerând că oricare dintre cele $n!$ permutări ale celor n chei este egal probabilă. (Exerciţiul 13.4-2 vă va cere să arătaţi că această noţiune este diferită de cea care presupune că orice arbore binar de căutare cu n chei are aceeaşi probabilitate de apariţie.) Scopul secţiunii de faţă este să demonstreze că înălţimea medie a unui arbore binar de căutare construit aleator având n chei este $O(\lg n)$.

Vom începe cu investigarea structurii arborilor binari de căutare care sunt construiţi folosind numai operaţii de inserare.

Lema 13.3 Notăm cu T arborele care rezultă prin inserarea a n chei distincte $k_1, k_2, \ldots, k_n$ (în ordine) într-un arbore binar de căutare iniţial vid. Cheia k_i este un strămoş al cheii k_j în T, pentru $1 \leq i < j \leq n$, dacă şi numai dacă:

$$k_i = \min\{k_l : 1 \leq l \leq i \text{ şi } k_l > k_j\}$$

sau

$$k_i = \max\{k_l : 1 \leq l \leq i \text{ şi } k_l < k_j\}.$$

Demonstraţie. $\Rightarrow$: Presupunem că k_i este un strămoş al lui k_j. Notăm cu T_i arborele care rezultă după ce au fost inserate în ordine cheile $k_1, k_2, \ldots, k_i$. Drumul de la rădăcină la nodul k_i în T_i este acelaşi cu drumul de la rădăcină la nodul k_i în T. De aici rezultă că dacă s-ar insera în T_i nodul k_j, acesta (k_j) ar deveni fie fiu stâng, fie fiu drept al nodului k_i. Prin urmare (vezi exerciţiul 13.2-6), k_i este fie cea mai mică cheie dintre $k_1, k_2, \ldots, k_i$ care este mai mare decât k_j, fie cea mai mare cheie dintre $k_1, k_2, \ldots, k_i$ care este mai mică decât k_j.

$\Leftarrow$: Presupunem că k_i este cea mai mică cheie dintre $k_1, k_2, \ldots, k_i$ care este mai mare decât k_j. (Cazul când k_i este cea mai mare cheie dintre $k_1, k_2, \ldots, k_i$ care este mai mică decât k_j se tratează simetric). Compararea cheii k_j cu oricare dintre cheile de pe drumul de la rădăcină la k_i în arborele T, produce aceleaşi rezultate ca şi compararea cheii k_i cu cheile respective. Prin urmare, pentru inserarea lui k_j se va parcurge drumul de la rădăcină la k_i, apoi k_j se va insera ca descendent al lui k_i. ∎

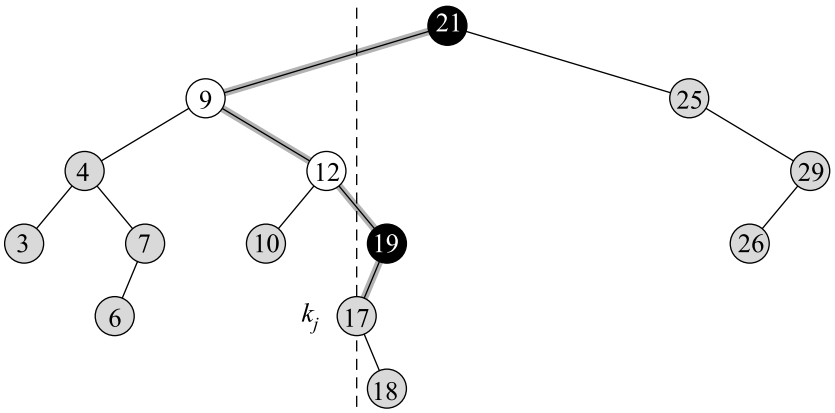
Următorul corolar al lemei 13.3 caracterizează precis adâncimea unei chei în funcţie de permutarea de intrare.

Corolarul 13.4 Fie T arborele care rezultă prin inserarea a n chei distincte $k_1, k_2, \ldots, k_n$ (în ordine) într-un arbore binar de căutare iniţial vid. Pentru o cheie k_j dată, unde $1 \leq j \leq n$, definim mulţimile:

$$G_j = \{k_i : 1 \leq i < j \text{ şi } k_l > k_i > k_j \text{ pentru toţi indicii } l < i \text{ cu } k_l > k_j\}$$

şi

$$L_j = \{k_i : 1 \leq i < j \text{ şi } k_l < k_i < k_j \text{ pentru toţi indicii } l < i \text{ cu } k_l < k_j\}\ .$$



(a)

chei	21	9	4	25	7	12	3	10	19	29	17	6	26	18
G'_j	21			25					19	29				
G_j	21								19					
L'_j		9	4		7	12	3	10						
L_j		9				12								

(b)

Figura 13.5 Ilustrarea celor două mulţimi G_j şi L_j care conţin cheile de pe drumul de la rădăcina unui arbore binar de căutare la cheia $k_j = 17$. **(a)** Nodurile care au cheile în G_j sunt haşurate cu negru, iar nodurile care au cheile în L_j sunt albe. Toate celelalte noduri sunt haşurate cu gris. Drumul în jos de la rădăcină la nodul cu cheia k_j este îngroşat. Cheile care sunt la stânga liniei îngroşate sunt mai mici decât k_j, iar cele de la dreapta acestei linii sunt mai mari decât k_j. Arborele se construieşte prin inserarea cheilor în ordinea dată de prima listă din **(b)**. Mulţimea $G'_j = \{21, 25, 19, 29\}$ conţine acele elemente care se inserează înainte de 17 şi sunt mai mari decât 17. Mulţimea $G_j = \{21, 19\}$ conţine acele elemente care modifică minimul elementelor din G'_j. Astfel, 21 este în G_j deoarece el este primul element din mulţime. Cheia 25 nu este în G_j deoarece este mai mare decât elementul minim curent, adică 21. Cheia 19 este în G_j deoarece ea este mai mică decât minimul curent care este 21 şi deci noul minim curent va fi 19. Cheia 29 nu este în G_j deoarece este mai mare decât elementul minim curent, adică 19. Structura mulţimilor L'_j şi L_j este simetrică.

Atunci cheile de pe drumul de la rădăcină la k_j sunt chiar cheile din $G_j \cup L_j$ iar adâncimea oricărei chei k_j din T este

$$d(k_j, T) = |G_j| + |L_j|.$$

■

Figura 13.5 ilustrează cele două mulţimi G_j şi L_j. G_j conţine toate cheile k_i inserate înainte de k_j astfel încât k_i este cea mai mică cheie din $k_1, k_2, \ldots, k_i$ care este mai mare decât k_j (structura lui L_j este simetrică). Pentru a înţelege mai bine mulţimea G_j, să încercăm să găsim o metodă de enumerare a elementelor sale. Dintre cheile $k_1, k_2, \ldots, k_{j-1}$ vom considera în ordine numai pe acelea care sunt mai mari decât k_j. Aceste chei formează mulţimea notată în figură cu G'_j.

Pe măsură ce inserăm cheile, vom reţine, în timp real, minimul. Mulţimea G_j va fi formată din acele chei care suprascriu minimul.

Pentru a facilita analiza, vom simplifica oarecum scenariul actual. Să presupunem că,într-o mulţime dinamică, se inserează pe rând n numere distincte, câte unul o dată. Dacă toate permutările numerelor respective sunt egal probabile, de câte ori se modifică în medie minimul mulţimii numerelor deja inserate? Pentru a răspunde la această întrebare, să presupunem că al i-lea număr inserat este k_i, pentru $i = 1, 2, \ldots, n$. Probabilitatea ca k_i să fie minimul primelor i numere este $1/i$, deoarece rangul (poziţia) lui k_i între primele i numere este egal probabil cu oricare dintre cele i ranguri posibile. Prin urmare, numărul mediu de modificări ale minimului mulţimii este

$$\sum_{i=1}^{n} \frac{1}{i} = H_n,$$

unde $H_n = \ln n + O(1)$ este al n-lea număr armonic (vezi relaţia (3.5) şi problema 6-2).

Prin urmare, ne aşteptăm ca numărul de modificări ale minimului să fie aproximativ egal cu $\ln n$, iar lema următoare arată că probabilitatea ca numărul să fie mult mai mare este foarte mică.

Lema 13.5 Fie $k_1, k_2, \ldots, k_n$ o permutare oarecare a unei mulţimi de n numere distincte şi fie $|S|$ variabila aleatoare care este cardinalul mulţimii

$$S = \{k_i : 1 \le i \le n \text{ şi } \ k_l > k_i \text{ pentru orice } \ l < i\} \ . \tag{13.1}$$

Atunci $Pr\{|S| \ge (\beta+1)H_n\} \le 1/n^2$, unde H_n este al n-lea număr armonic, iar $\beta \approx 4,32$ verifică ecuaţia $(\ln \beta - 1)\beta = 2$.

Demonstraţie. Putem considera că determinarea cardinalului mulţimii S se face prin n încercări Bernoulli, având succes la a i-a încercare dacă k_i este mai mic decât elementele $k_1, k_2, \ldots, k_{i-1}$. Probabilitatea de a avea succes la a i-a încercare este $1/i$. Încercările sunt independente deoarece probabilitatea ca k_i să fie minimul dintre elementele $k_1, k_2, \ldots, k_i$ este independentă de ordonarea relativă a mulţimii $k_1, k_2, \ldots, k_{i-1}$.

Putem folosi teorema 6.6 pentru a mărgini probabilitatea ca $|S| \ge (\beta+1)H_n$. Valoarea medie pentru $|S|$ este $\mu = H_n \ge \ln n$. Deoarece $\beta > 1$, din teorema 6.6 rezultă

$$\begin{aligned} Pr\{|S| \ge (\beta+1)H_n\} &= Pr\{|S| - \mu \ge \beta H_n\} \le \left(\frac{eH_n}{\beta H_n}\right)^{\beta H_n} = e^{(1-\ln\beta)\beta H_n} \\ &\le e^{-(\ln\beta - 1)\beta \ln n} = n^{-(\ln\beta-1)\beta} = 1/n^2 \ , \end{aligned}$$

în care am folosit definiţia lui β. ∎

Acum avem instrumente pentru a mărgini înălţimea unui arbore binar de căutare construit aleator.

Teorema 13.6 Înălţimea medie a unui arbore binar de căutare construit aleator cu n chei distincte este $O(\lg n)$.

Demonstraţie. Fie $k_1, k_2, \ldots, k_n$ o permutare oarecare a celor n chei şi fie T arborele binar de căutare care rezultă prin inserarea cheilor în ordinea specificată pornind de la un arbore iniţial vid. Vom discuta prima dată probabilitatea ca adâncimea $d(k_j, T)$ a unei chei date k_j să fie cel puţin t, pentru o valoare t arbitrară. Conform caracterizării adâncimii $d(k_j, T)$ din corolarul 13.4, dacă adâncimea lui k_j este cel puţin t, atunci cardinalul uneia dintre cele două mulţimi G_j şi L_j trebuie să fie cel puţin $t/2$. Prin urmare,

$$Pr\{d(k_j, T) \geq t\} \leq Pr\{|G_j| \geq t/2\} + Pr\{|L_j| \geq t/2\} \ . \tag{13.2}$$

Să examinăm la început $Pr\{|G_j| \geq t/2\}$. Avem

$$\begin{aligned} Pr\{|G_j| &\geq t/2\} \\ &= Pr\{|\{k_i : 1 \leq i < j \text{ şi } k_l > k_i > k_j \text{ pentru orice } l < i\}| \geq t/2\} \\ &\leq Pr\{|\{k_i : i \leq n \text{ şi } k_l > k_i \text{ pentru orice } l < i\}| \geq t/2\} \\ &= Pr\{|S| \geq t/2\} \ , \end{aligned}$$

unde S este definit în relaţia (13.1). În sprijinul acestei afirmaţii, să observăm că probabilitatea nu va descreşte dacă vom extinde intervalul de variaţie al lui i de la $i < j$ la $i \leq n$, deoarece prin extindere se vor adăuga elemente noi la mulţime. Analog, probabilitatea nu va descreşte dacă se renunţă la condiţia $k_i > k_j$, deoarece prin aceasta se înlocuieşte o permutare a (de regulă) mai puţin de n elemente (şi anume acele chei k_i care sunt mai mari decât k_j) cu o altă permutare oarecare de n elemente.

Folosind o argumentare similară, putem demonstra că

$$Pr\{|L_j| \geq t/2\} \leq Pr\{|S| \geq t/2\} \ ,$$

şi apoi, folosind inegalitatea (13.2) obţinem:

$$Pr\{d(k_j, T) \geq t\} \leq 2Pr\{|S| \geq t/2\} \ .$$

Dacă alegem $t = 2(\beta + 1)H_n$, unde H_n este al n-lea număr armonic iar $\beta \approx 4.32$ verifică ecuaţia $(\ln \beta - 1)\beta = 2$, putem aplica lema 13.5 pentru a concluziona că

$$Pr\{d(k_j, T) \geq 2(\beta + 1)H_n\} \leq 2Pr\{|S| \geq (\beta + 1)H_n\} \leq 2/n^2 \ .$$

Deoarece discutăm despre un arbore binar de căutare construit aleator şi cu cel mult n noduri, probabilitatea ca adâncimea *oricăruia* dintre noduri să fie cel puţin $2(\beta + 1)H_n$ este, folosind inegalitatea lui Boole (6.22), de cel mult $n(2/n^2) = 2/n$. Prin urmare, în cel puţin $1 - 2/n$ din cazuri, înălţimea arborelui binar de căutare construit aleator este mai mică decât $2(\beta + 1)H_n$ şi în cel mult $2/n$ din cazuri înălţimea este cel mult n. În concluzie, înălţimea medie este cel mult $(2(\beta + 1)H_n)(1 - 2/n) + n(2/n) = O(\lg n)$. ∎

Exerciţii

13.4-1 Descrieţi un arbore binar de căutare cu n noduri astfel ca adâncimea medie a unui nod în arbore să fie $\Theta(\lg n)$ iar înălţimea arborelui să fie $\omega(\lg n)$. Cât de mare poate fi înălţimea unui arbore binar de căutare având n noduri dacă adâncimea medie a unui nod este $\Theta(\lg n)$?

13.4-2 Arătaţi că noţiunea de arbore binar de căutare având n chei ales arbitrar (suntem în situaţia în care oricare arbore binar de căutare având n chei este egal probabil a fi ales), este diferită de noţiunea de arbore binar de căutare având n noduri construit aleator, dată în această secţiune. (*Indicaţie:* Analizaţi toate posibilităţile pentru $n = 3$.)

13.4-3 $\star$ Fiind dată o constantă $r \geq 1$, determinaţi pe t astfel încât probabilitatea ca înălţimea arborelui binar de căutare construit aleator să fie cel puţin tH_n, să fie mai mică decât $1/n^r$.

13.4-4 $\star$ Considerăm algoritmul QUICKSORT-ALEATOR ce are ca date de intrare un şir de n numere. Demonstraţi că pentru orice constantă $k > 0$ toate cele $n!$ permutări, mai puţin $O(1/n^k)$ dintre ele, se execută într-un timp $O(n \lg n)$.

Probleme

13-1 Arbori binari de căutare cu chei egale
Cheile egale pun probleme implementării arborilor binari de căutare.

a. Care este performanţa asimptotică a algoritmului ARBORE-INSEREAZĂ când acesta se foloseşte pentru a insera n elemente cu chei identice într-un arbore binar de căutare, iniţial vid?

Ne propunem să îmbunătăţim algoritmul ARBORE-INSEREAZĂ testând înainte de linia 5 dacă $cheie[z] = cheie[x]$ şi înainte de linia 12 dacă $cheie[z] = cheie[y]$. Dacă are loc egalitatea, se va implementa una dintre strategiile descrise în continuare. Găsiţi pentru fiecare strategie performanţa asimptotică a inserării de n elemente având chei identice într-un arbore binar de căutare iniţial vid (Strategiile sunt descrise pentru linia 5, în care se compară cheile lui z şi x. Pentru a obţine strategiile pentru linia 12, înlocuiţi x cu y).

b. Se foloseşte un comutator boolean $b[x]$ pentru nodul x, şi se setează x la *stânga*$[x]$ sau $dreapta[x]$ în funcţie de valoarea lui $b[x]$, care va lua alternativ valorile FALS respectiv ADEVĂRAT la fiecare vizitare a nodului în ARBORE-INSEREAZĂ.

c. Se memorează o listă de noduri având cheile egale cu cheia din x şi se inserează z în acea listă.

d. Se atribuie aleator lui x fie *stânga*$[x]$, fie $dreapta[x]$. (Stabiliţi performanţa în cazul cel mai defavorabil şi deduceţi intuitiv performanţa în cazul mediu).

13-2 Arbori cu rădăcină
Fiind date două şiruri de caractere $a = a_0 a_1 \ldots a_p$ şi $b = b_0 b_1 \ldots b_q$, în care fiecare a_i şi fiecare b_j sunt caractere dintr-o mulţime ordonată de caractere, spunem că şirul de caractere a este ***mai mic lexicografic*** decât şirul de caractere b dacă se verifică una dintre condiţiile următoare:

1. există un întreg j, $0 \leq j \leq \min(p, q)$, astfel încât $a_i = b_i$ pentru orice $i = 0, 1, \ldots j - 1$ şi $a_j < b_j$, sau
2. $p < q$ şi $a_i = b_i$ pentru orice $i = 0, 1, \ldots p$.

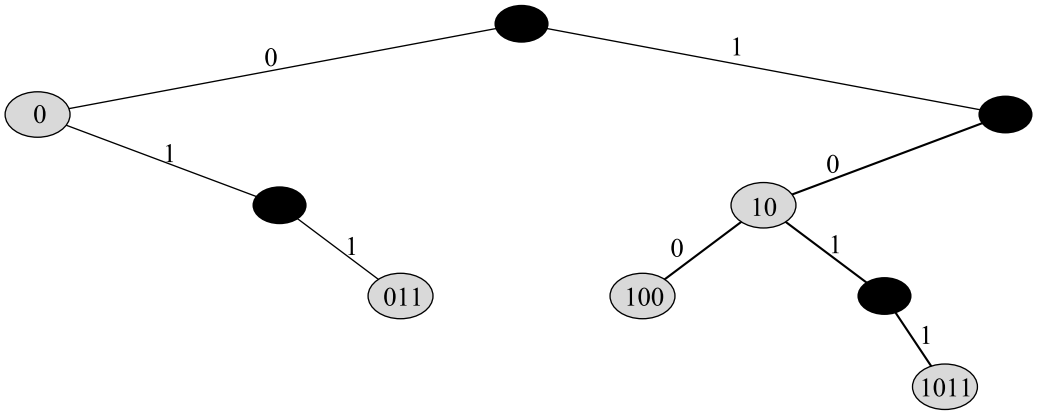


Figura 13.6 Un arbore cu rădăcină care memorează şirurile de biţi 1011, 10, 011, 100 şi 0. Fiecare cheie a unui nod se poate determina prin traversarea drumului de la rădăcină la nodul respectiv. Prin urmare nu este nevoie să memorăm cheile în noduri; cheile sunt afişate numai în scop ilustrativ. Nodurile sunt haşurate cu negru dacă cheile care le corespund nu sunt în arbore; astfel de noduri sunt prezente numai pentru a stabili un drum spre alte noduri.

De exemplu, dacă a şi b sunt şiruri de biţi, atunci $10100 < 10110$ pe baza regulii 1 (punând $j = 3$), iar $10100 < 101000$ pe baza regulii 2. Această ordonare este similară celei folosite în dicţionarele limbii engleze (şi nu numai).

Structura de date ***arbore cu rădăcină*** ilustrată în figura 13.6 memorează şirurile de biţi 1011, 10, 011, 100 şi 0. Dacă dorim să căutăm o cheie $a = a_0 a_1 \ldots a_p$ şi ne aflăm într-un nod de adâncime i, vom merge la stânga dacă $a_i = 0$ şi la dreapta dacă $a_i = 1$. Fie S o mulţime de şiruri de caractere binare distincte care au suma lungimilor egală cu n. Arătaţi cum se foloseşte arborele ce rădăcină pentru a sorta lexicografic S într-un timp $\Theta(n)$. De exemplu, în figura 13.6 rezultatul sortării va fi şirul 0, 011, 10, 100, 1011.

13-3 Adâncimea medie a unui nod într-un arbore binar de căutare construit aleator
În această problemă vom demonstra că adâncimea medie a unui nod într-un arbore binar de căutare având n noduri construit aleator este $O(\lg n)$. Cu toate că acest rezultat este mai slab decât cel dat în teorema 13.6, tehnica pe care o vom folosi relevă o similaritate surprinzătoare între construirea unui arbore binar de căutare şi execuţia algoritmului QUICKSORT-ALEATOR din secţiunea 8.3. Ne reamintim din capitolul 5 că lungimea drumului interior $P(T)$ al unui arbore binar T este suma adâncimilor nodurilor x, notată cu $d(x, T)$ peste toate nodurile x din T.

a. Demonstraţi că adâncimea medie a unui nod în T este

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$

Acum dorim să arătăm că valoarea medie pentru $P(T)$ este $O(n \lg n)$.

b. Notăm cu T_L şi respectiv T_R subarborele stâng, respectiv subarborele drept al arborelui T. Demonstraţi că dacă T are n noduri, atunci

$$P(T) = P(T_L) + P(T_R) + n - 1.$$

c. Notăm cu $P(n)$ lungimea medie a drumului interior al unui arbore binar de căutare construit aleator având n noduri. Arătaţi că

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) \ .$$

d. Arătaţi că $P(n)$ se poate rescrie astfel:

$$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n).$$

e. Reamintindu-vă analiza versiunii aleatoare a algoritmului QUICKSORT, demonstraţi că $P(n) = O(n \lg n)$.

La fiecare apelare recursivă a algoritmului QUICKSORT, alegem aleator un element pivot pentru a partiţiona mulţimea elementelor de sortat. Fiecare nod al unui arbore binar de căutare împarte în două mulţimea de elemente care aparţin subarborelui cu rădăcina în respectivul nod.

f. Descrieţi o implementare a algoritmului QUICKSORT în care comparările efectuate pentru a sorta o mulţime de elemente sunt exact aceleaşi care se folosesc pentru a insera elementele într-un arbore binar de căutare. (Ordinea în care se fac comparările poate să difere, însă trebuie făcute exact aceleaşi comparări.)

13-4 Numărul de arbori binari distincţi

Notăm cu b_n numărul de arbori binari distincţi având n noduri. În această problemă veţi găsi o formulă pentru b_n şi o estimare asimptotică.

a. Arătaţi că $b_0 = 1$ şi că, pentru $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$

b. Fie $B(x)$ funcţia generatoare

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

(vezi problema 4-6 pentru definiţia funcţiilor generatoare). Arătaţi că $B(x) = xB(x)^2 + 1$ şi de aici că

$$B(x) = \frac{1}{2x}(1 - \sqrt{1 - 4x}).$$

Dezvoltarea în serie Taylor a lui $f(x)$ în vecinătatea punctului $x = a$ este dată de

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k,$$

unde $f^{(k)}(x)$ este derivata de ordinul k a lui f în punctul x.

c. Arătaţi că

$$b_n = \frac{1}{n+1}\binom{2n}{n}$$

(al n-lea ***număr Catalan***) folosind dezvoltarea în serie Taylor a expresiei $\sqrt{1-4x}$ în vecinătatea lui $x = 0$. (Dacă doriţi, puteţi folosi generalizarea dezvoltării binomiale (6.5) pentru exponenţi neîntregi n în locul dezvoltării în serie Taylor, în care pentru orice număr real n şi orice întreg k, coeficientul binomial $\binom{n}{k}$ va fi interpretat ca $n(n-1)\ldots(n-k+1)/k!$ dacă $k \geq 0$ şi 0 în caz contrar.)

d. Arătaţi că

$$b_n = \frac{4^n}{\sqrt{\pi}n^{3/2}}(1 + O(1/n)).$$

Note bibliografice

Knuth [123] conţine o discuţie pertinentă a arborilor binari de căutare simpli şi de asemenea a numeroase variante ale acestora. Se pare că arborii binari de căutare au fost descoperiţi independent de mai mulţi cercetători la sfârşitul anilor ’50.

14 Arbori roşu-negru

Capitolul 13 a arătat că un arbore binar de căutare de înălţime h poate implementa oricare dintre operaţiile de bază pe mulţimi dinamice – precum CAUTĂ, PREDECESOR, SUCCESOR, MINIM, MAXIM, INSEREAZĂ şi ŞTERGE – într-un timp $O(h)$. Prin urmare, operaţiile pe mulţimi sunt rapide dacă înălţimea arborelui de căutare este mică; dacă însă arborele are înălţimea mare, performanţele operaţiilor s-ar putea să nu fie mai bune decât cele ale listelor înlănţuite. Arborii roşu-negru sunt un caz particular de arbori de căutare care sunt "echilibraţi" în ideea garantării unui timp $O(\lg n)$ pentru cazul cel mai defavorabil al execuţiei operaţiilor de bază pe mulţimi dinamice.

14.1. Proprietăţile arborilor roşu-negru

Un ***arbore roşu-negru*** este un arbore binar de căutare care are un bit suplimentar pentru memorarea fiecărui nod: ***culoarea*** acestuia, care poate fi ROŞU sau NEGRU. Prin restrângerea modului în care se colorează nodurile pe orice drum de la rădăcină la o frunză, arborii roşu-negru garantează că nici un astfel de drum nu este mai lung decât dublul lungimii oricărui alt drum, deci că arborele este aproximativ ***echilibrat***.

Fiecare nod al arborelui conţine astfel câmpurile *culoare*, *cheie*, *stânga*, *dreapta* şi *p*. Dacă fiul sau părintele unui nod nu există, câmpul pointer corespunzător din nod are valoarea NIL. Vom considera că aceste valori NIL sunt pointeri la noduri (frunze) externe arborelui binar de căutare şi că nodurile obişnuite, care conţin chei, sunt noduri interne ale arborelui.

Un arbore binar de căutare este arbore roşu-negru dacă el îndeplineşte următoarele ***proprietăţi roşu-negru***:

1. Fiecare nod este fie roşu, fie negru.
2. Fiecare frunză (NIL) este neagră.
3. Dacă un nod este roşu, atunci ambii fii ai săi sunt negri.
4. Fiecare drum simplu de la un nod la un descendent care este frunză conţine acelaşi număr de noduri negre.

În figura 14.1 este prezentat un exemplu de arbore roşu-negru.

Numărul de noduri negre pe orice drum de la un nod (fără a include în numărare şi nodul) în jos spre o frunză se va numi ***înălţime neagră*** a nodului şi se va nota cu $\mathrm{bh}(x)$. Noţiunea de înălţime neagră este bine definită pe baza proprietăţii 4, conform căreia toate drumurile de la nod în jos au acelaşi număr de noduri negre. Vom defini înălţimea neagră a unui arbore roşu-negru ca fiind înălţimea neagră a rădăcinii sale.

Lema următoare va arăta de ce arborii roşu-negru sunt arbori de căutare eficienţi.

Lema 14.1 Un arbore roşu-negru cu n noduri interne are înălţimea mărginită superior de $2\lg(n+1)$.

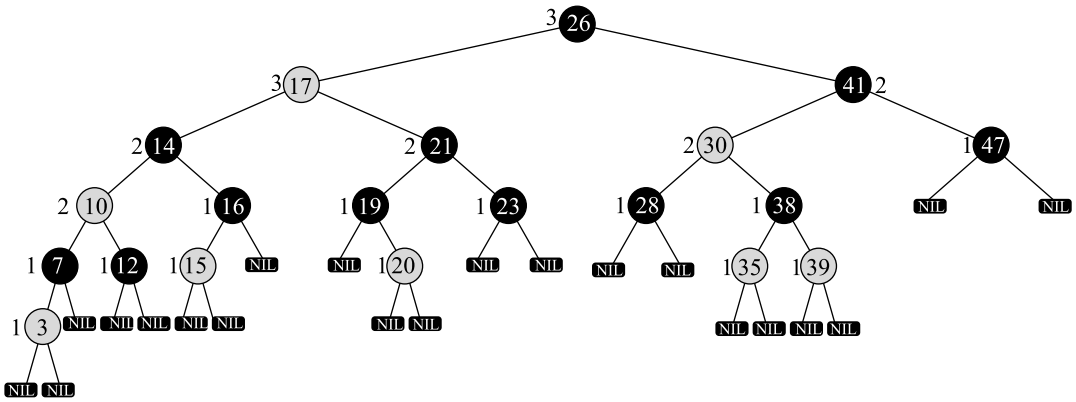


Figura 14.1 Un arbore roşu-negru în care nodurile negre sunt înnegrite, iar nodurile roşii sunt gri. Fiecare nod dintr-un asemenea arbore este fie roşu fie negru, fiecare frunză (NIL) este neagră, fiii unui nod roşu sunt ambii negri, iar oricare drum simplu de la un nod la un descendent frunză conţine acelaşi număr de noduri negre. Fiecare nod diferit de NIL este marcat cu un număr ce reprezintă înălţimea neagră a nodului; nodurile NIL au înălţimea neagră egală cu 0.

Demonstraţie. Vom arăta la început că subarborele care are ca rădăcină un nod oarecare x conţine cel puţin $2^{\mathrm{bh}(x)} - 1$ noduri interne. Această afirmaţie va fi demonstrată prin inducţie după înălţimea nodului x. Dacă înălţimea lui x este 0, atunci x trebuie să fie frunză (NIL), iar subarborele cu rădăcina x conţine cel puţin $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ noduri interne. Pentru pasul de inducţie vom considera un nod x care are înălţimea pozitivă şi este nod intern cu doi fii. Fiecare fiu are înălţimea neagră egală fie cu $\mathrm{bh}(x)$, fie cu $\mathrm{bh}(x) - 1$, în funcţie de culoarea sa (roşu, respectiv negru). Deoarece înălţimea unui fiu al lui x este mai mică decât înălţimea lui x, putem aplica ipoteza inducţiei pentru a demonstra că fiecare fiu are cel puţin $2^{\mathrm{bh}(x)-1} - 1$ noduri interne. Prin urmare, subarborele cu rădăcina x conţine cel puţin $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 1 = 2^{\mathrm{bh}(x)} - 1$ noduri interne, ceea ce demonstrează afirmaţia de mai sus.

Pentru a termina demonstraţia lemei, notăm cu h înălţimea arborelui. Conform proprietăţii 3, cel puţin jumătate dintre nodurile de pe orice drum simplu de la rădăcină la o frunză, fără a număra şi rădăcina, trebuie să fie colorate cu negru. Prin urmare, înălţimea neagră a rădăcinii trebuie să fie cel puţin $h/2$; de aici rezultă

$$n \geq 2^{h/2} - 1.$$

Mutând pe 1 în membrul stâng şi logaritmând ambii membri, obţinem că $\lg(n+1) \geq h/2$ sau că $h \leq 2\lg(n+1)$. ∎

O consecinţă imediată a acestei leme este că operaţiile pe mulţimi dinamice CAUTĂ, MINIM, MAXIM, SUCCESOR şi PREDECESOR se pot implementa pe arborii roşu-negru într-un timp $O(\lg n)$, deoarece pe un arbore binar de căutare de înălţime h aceste operaţii se pot executa într-un timp $O(h)$ (după cum s-a arătat în capitolul 13) şi deoarece orice arbore roşu-negru cu n noduri este un arbore de căutare cu înălţimea $O(\lg n)$. Cu toate că algoritmii ARBORE-INSEREAZĂ şi ARBORE-ŞTERGE prezentaţi în capitolul 13 se execută într-un timp $O(\lg n)$ pe un arbore roşu-negru, ei nu sprijină direct operaţiile INSEREAZĂ şi ŞTERGE pe mulţimi dinamice, deoarece ei nu garantează că arborele binar de căutare rezultat prin aplicarea lor va rămâne

arbore roşu-negru. În secţiunile 14.3 şi 14.4 vom vedea că şi aceste două operaţii se pot executa într-un timp $O(\lg n)$.

Exerciţii

14.1-1 Trasaţi arborele binar de căutare complet de înălţime 3 având cheile $\{1, 2, \ldots, 15\}$. Adăugaţi frunze NIL şi coloraţi nodurile în trei moduri diferite astfel încât înălţimile negre ale arborilor roşu-negru rezultaţi să fie 2, 3 şi respectiv 4.

14.1-2 Presupunem că rădăcina unui arbore roşu-negru are culoarea roşu. Dacă îi schimbăm culoarea în negru, noul arbore este tot roşu-negru?

14.1-3 Arătaţi că cel mai lung drum simplu de la un nod x, dintr-un arbore roşu-negru, la o frunză descendent are lungimea cel mult dublul lungimii celui mai scurt drum simplu de la nodul x la o frunză descendent.

14.1-4 Care este cel mai mare număr posibil de noduri interne într-un arbore roşu-negru cu înălţimea neagră k? Dar cel mai mic?

14.1-5 Descrieţi un arbore roşu-negru având n chei care are cel mai mare raport posibil dintre numărul de noduri interne roşii şi numărul de noduri interne negre. Care este acest raport? Care arbore are cel mai mic raport, şi care este acest raport?

14.2. Rotaţii

Executând pe arbori roşu-negru având n chei operaţiile defimite pe arbori de căutare ARBORE-INSEREAZĂ şi ARBORE-ŞTERGE va consuma un timp de $O(\lg n)$. Deoarece ele modifică arborele, se poate întâmpla ca rezultatul efectuării lor să încalce proprietăţile roşu-negru enumerate în secţiunea 14.1. Pentru a reface aceste proprietăţi, atât culorile unora dintre nodurile arborelui cât şi structura pointerilor trebuie modificate.

Structura pointerilor se modifică prin ***rotaţie***, operaţie locală într-un arbore de căutare ce conservă ordonarea cheilor în inordine. Figura 14.2 ilustrează cele două tipuri de rotaţii: rotaţii la stânga şi rotaţii la dreapta. Când se efectuează o rotaţie la stânga pe un nod x, presupunem că fiul drept al acestuia, y, este diferit de NIL. Rotaţia la stânga "pivotează" în jurul legăturii de la x la y. Ea transformă pe y în rădăcină a subarborelui, pe x în fiu stâng al lui y şi pe fiul stâng al lui y în fiu drept al lui x.

Pseudocodul pentru ROTEŞTE-STÂNGA presupune că $dreapta[x] \neq$ NIL.

Figura 14.3 ilustrează modul de operare al algoritmului ROTEŞTE-STÂNGA. Codul pentru ROTEŞTE-DREAPTA este similar. Atât ROTEŞTE-STÂNGA cât şi ROTEŞTE-DREAPTA se execută în timp $O(1)$. Prin rotaţie se modifică numai pointerii; toate celelalte câmpuri ale nodului rămân aceleaşi.

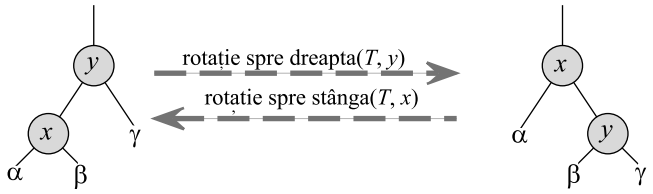


Figura 14.2 Operaţii de rotaţie pe un arbore binar de căutare. Operaţia ROTEŞTE-DREAPTA(T, y) transformă configuraţia de două noduri din stânga în configuraţia prezentată în dreapta prin modificarea unui număr constant de pointeri. Configuraţia din dreapta se poate transforma în configuraţia din stânga prin operaţia inversă ROTEŞTE-STÂNGA(T, x). Cele două noduri pot să apară oriunde într-un arbore binar de căutare. Literele α, β şi γ reprezintă subarbori arbitrari. Operaţia de rotaţie conservă ordonarea cheilor în inordine: cheile din α sunt înaintea lui $cheie[x]$, care este înaintea cheilor din β, care la rândul lor sunt mai mici decât $cheie[y]$ şi care, în sfârşit, este mai mică decât cheile din γ.

ROTEŞTE-STÂNGA(T, x)

```
 1: y ← dreapta[x]              ▷Setează y.
 2: dreapta[x] ←stânga[y]       ▷Transformă subarborele stâng al lui y în subarbore drept al lui x.
 3: dacă stânga[y] ≠ NIL atunci
 4:    p[stânga[y]] ← x
 5: p[y] ← p[x]                 ▷Leagă părintele lui x de y.
 6: dacă p[x] = NIL atunci
 7:    rădăcină[T] ← y
 8: altfel
 9:    dacă x =stânga[p[x]] atunci
10:       stânga[p[x]] ← y
11:    altfel
12:       dreapta[p[x]] ← y
13: stânga[y] ← x               ▷Pune pe x ca fiu stâng al lui y.
14: p[x] ← y
```

Exerciţii

14.2-1 Trasaţi arborele roşu-negru care rezultă după apelul procedurii ARBORE-INSEREAZĂ aplicată arborelui din figura 14.1, pentru cheia 36. Dacă nodul inserat este colorat cu roşu, va fi arborele rezultat un arbore roşu-negru? Dar dacă este colorat cu negru?

14.2-2 Scrieţi pseudocodul pentru ROTEŞTE-DREAPTA.

14.2-3 Fie a, b şi c noduri arbitrare în subarborii α, β şi respectiv γ din arborele plasat în partea stângă a figurii 14.2. Cum se modifică adâncimile nodurilor a, b şi c când se efectuează o rotaţie la dreapta pe nodul y din figură?

14.2-4 Arătaţi că orice arbore binar de căutare având n noduri poate fi transformat în orice alt arbore binar de căutare având n noduri, folosind $O(n)$ rotaţii. (*Indicaţie:* Arătaţi mai întâi că sunt suficiente cel mult $n - 1$ rotaţii la dreapta pentru a transforma arborele într-un lanţ spre dreapta).

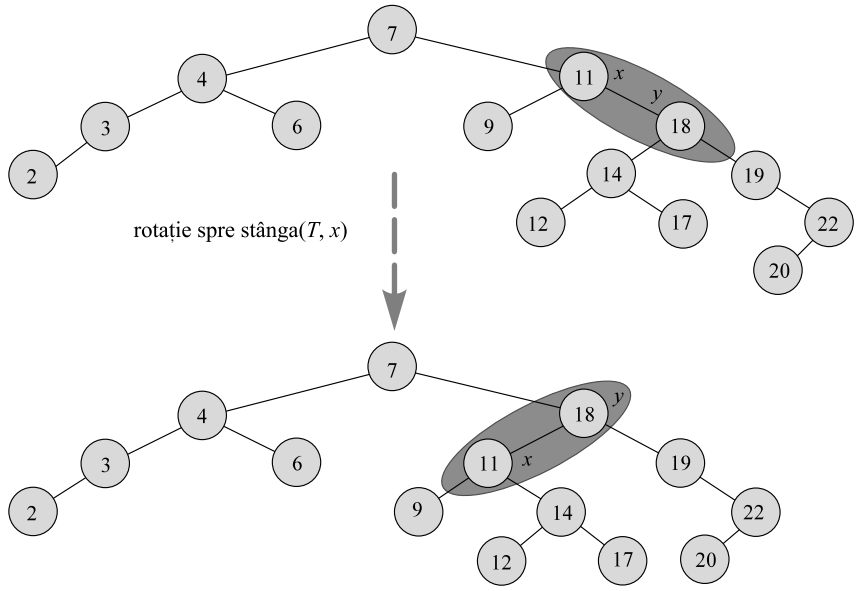


Figura 14.3 Un exemplu care ilustrează modul în care procedura ROTEŞTE-STÂNGA(T, x) modifică un arbore binar de căutare. Frunzele NIL sunt omise. Traversarea în inordine aplicată arborelui de intrare şi arborelui modificat produc aceeaşi listă de valori de chei.

14.3. Inserarea

Inserarea unui nod într-un arbore roşu-negru având n noduri se poate realiza într-un timp $O(\lg n)$. Vom utiliza la început procedura ARBORE-INSEREAZĂ (secţiunea 13.3) pentru a insera un nod x în arborele T considerând că T este un arbore binar de căutare obişnuit şi apoi îl vom colora pe x cu roşu. Pentru a garanta conservarea proprietăţilor roşu-negru, vom reface arborele rezultat prin recolorarea nodurilor şi efectuarea de rotaţii. Majoritatea codului algoritmului RN-INSEREAZĂ gestionează diversele cazuri ce pot apare în timpul refacerii proprietăţilor roşu-negru pentru arborele rezultat în urma execuţiei algoritmului ARBORE-INSEREAZĂ.

Codul algoritmului RN-INSEREAZĂ este mai puţin complicat decât arată şi îl vom examina în trei paşi majori. La început vom determina ce încălcări ale proprietăţilor roşu-negru se introduc în liniile 1–2 când se inserează în arbore nodul x şi apoi acesta se colorează cu roşu. Pe urmă vom examina motivaţia principală a ciclului **cât timp** din liniile 3–19. În sfârşit, vom explora fiecare dintre cele trei cazuri în care se subdivide corpul ciclului **cât timp** şi vom vedea cum sunt atinse deziderateleate. Figura 14.4 ilustrează modul de funcţionare al algoritmului RN-INSEREAZĂ pe un arbore roşu-negru eşantion.

Care dintre proprietăţile roşu-negru se pot încălca după execuţia liniilor 1–2? În mod sigur proprietatea 1 se conservă, la fel şi proprietatea 2, deoarece noul nod roşu inserat are fiii setaţi pe NIL. Proprietatea 4, care afirmă că numărul de noduri colorate cu negru este acelaşi pe orice drum ce pleacă dintr-un nod dat spre frunze, se păstrează de asemenea, deoarece nodul x înlocuieşte un nod NIL (negru), iar nodul x este roşu şi cu fiii coloraţi cu negru. Prin urmare, singura proprietate

care ar putea fi distrusă este proprietatea 3, care afirmă că un nod colorat cu roşu nu poate avea un fiu colorat şi el cu roşu. Mai exact, proprietatea 3 este încălcată dacă părintele lui x este colorat cu roşu, deoarece x este el însuşi colorat astfel în linia 2. Figura 14.4(a) ilustrează acest caz după ce nodul x a fost inserat şi colorat cu roşu.

```
RN-INSEREAZĂ(T, x)
 1: ARBORE-INSEREAZĂ(T, x)
 2: culoare[x] ← ROŞU
 3: cât timp x ≠rădăcină[T] şi culoare[p[x]]=ROŞU execută
 4:   dacă p[x] =stânga[p[p[x]]] atunci
 5:     y ← dreapta[p[p[x]]]
 6:     dacă culoare[y] = ROŞU atunci
 7:       culoare[p[x]] ← NEGRU              ▷Cazul 1
 8:       culoare[y] ← NEGRU                 ▷Cazul 1
 9:       culoare[p[p[x]]] ← ROŞU            ▷Cazul 1
10:       x ← p[p[x]]                        ▷Cazul 1
11:     altfel
12:       dacă x = dreapta[p[x]] atunci
13:         x ← p[x]                         ▷Cazul 2
14:         ROTEŞTE-STÂNGA(T, x)             ▷Cazul 2
15:       culoare[p[x]] ← NEGRU              ▷Cazul 3
16:       culoare[p[p[x]]] ← ROŞU            ▷Cazul 3
17:       ROTEŞTE-DREAPTA(T, p[p[x]])        ▷Cazul 3
18:   altfel
19:       (La fel ca în clauza atunci interschimbând "dreapta" cu "stânga")
20: culoare[rădăcină[T]] ← NEGRU
```

Scopul principal al ciclului **cât timp** din liniile 3–19 este să mute în sus în arbore nodul care încalcă proprietatea 3 şi în acelaşi timp să păstreze ca invariant proprietatea 4. La începutul fiecărei iteraţii a ciclului, x va referi un nod colorat cu roşu şi care are părintele colorat cu roşu - singura încălcare a proprietăţilor roşu-negru din arbore. Există două rezultate posibile ale fiecărei iteraţii din ciclu: ori pointerul x se mută în sus în arbore, ori se efectuează anumite rotaţii şi ciclul se încheie.

Există de fapt şase cazuri care trebuie considerate în ciclul **cât timp**, însă trei dintre ele sunt simetrice celorlalte trei, în funcţie de cum părintele $p[x]$ al lui x este fiul stâng sau fiul drept al bunicului $p[p[x]]$ al lui x, lucru care se determină în linia 4. Pseudocodul conţine numai cazul în care $p[x]$ este fiul stâng al părintelui său, $p[p[x]]$. S-a făcut o presupunere importantă, că rădăcina arborelui este colorată cu negru - o proprietate care se garantează în linia 20 ori de câte ori algoritmul se termină - deci $p[x]$ nu este rădăcina arborelui şi $p[p[x]]$ există.

Cazul 1 se deosebeşte de cazurile 2 şi 3 prin culoarea fratelui părintelui lui x ("unchiul" lui x). Linia 5 face pe y să refere unchiul $dreapta[p[p[x]]]$ al lui x şi apoi se face un test în linia 6. Dacă y este colorat cu roşu, se execută cazul 1. Altfel, se vor executa cazurile 2 sau 3. În toate cele trei cazuri, bunicul $p[p[x]]$ al lui x este negru, deoarece părintele $p[x]$ al lui x este roşu iar proprietatea 3 este încălcată numai între x şi $p[x]$.

Prelucrările corespunzătoare cazului 1 (liniile 7–10) sunt ilustrate în figura 14.5. Acest caz se execută când atât $p[x]$ cât şi x sunt coloraţi cu roşu. Deoarece $p[p[x]]$ este colorat cu negru,

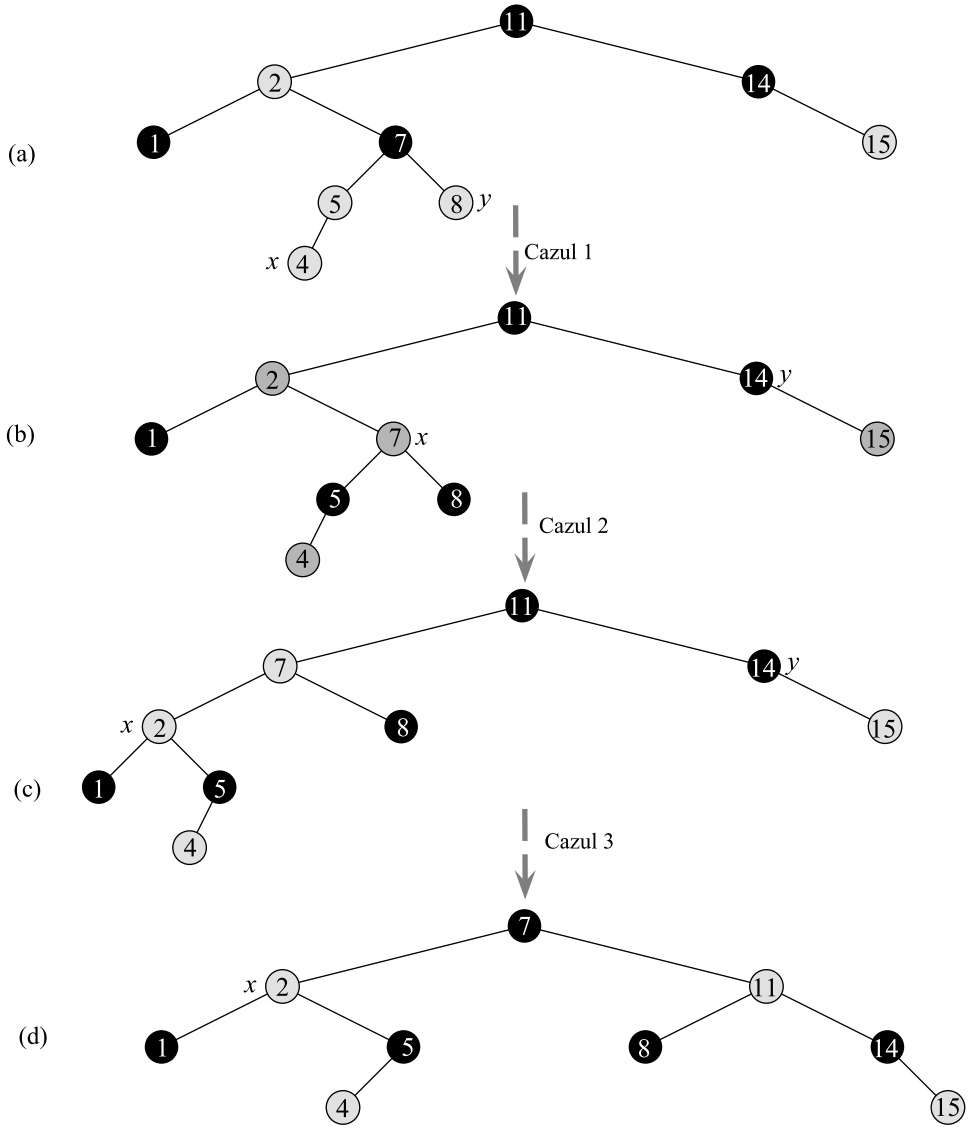


Figura 14.4 Modul de operare al algoritmului RN-Inserează. **(a)** Nodul x după inserare. Deoarece x şi părintele său $p[x]$ sunt ambii coloraţi cu roşu, proprietatea 3 nu este verificată. Deoarece unchiul y al lui x este tot roşu, se poate folosi cazul 1 din cod. Nodurile sunt recolorate şi pointerul x este mutat în sus în arbore, rezultând arborele prezentat în **(b)**. Din nou, x şi părintele său $p[x]$ sunt ambii coloraţi cu roşu, însă unchiul y al lui x este de data aceasta colorat cu negru. Deoarece x este fiul drept al lui $p[x]$, se poate folosi cazul 2. Se efectuează o rotaţie la stânga şi arborele care rezultă este prezentat în **(c)**. Acum x este fiul stâng al părintelui său şi se va folosi cazul 3. Se efectuează o rotaţie la dreapta care va produce arborele din **(d)**, care este un arbore roşu-negru valid.

atât $p[x]$ cât şi x se pot colora cu negru, rezolvând prin urmare conflictul că $p[x]$ şi x sunt ambii coloraţi cu roşu, şi apoi vom colora pe $p[p[x]]$ cu roşu, conservând astfel proprietatea 4. Singura problemă care poate să apară este ca $p[p[x]]$ să aibă la rândul său un părinte colorat cu roşu; prin urmare trebuie să repetăm ciclul **cât timp** cu $p[p[x]]$ pe postul noului nod x.

În cazurile 2 şi 3, culoarea unchiului y al lui x este neagră. Cele două cazuri se deosebesc prin aceea că x este fiul drept, respectiv fiul stâng al lui $p[x]$. Liniile 13–14 corespund cazului 2, care este ilustrat împreună cu cazul 3 în figura 14.6. În cazul 2, nodul x este fiul drept al părintelui său. Prin folosirea unei rotaţii la stânga, acest caz se transformă în cazul 3 (liniile 15–17), în care nodul x este fiul stâng al părintelui său. Deoarece atât x cât şi $p[x]$ sunt coloraţi cu roşu, rotaţia nu afectează nici înălţimea neagră a nodurilor, nici proprietatea 4. Indiferent dacă ajungem la cazul 3 direct sau prin cazul 2, acum unchiul y al lui x este colorat cu negru, deoarece altfel am fi executat cazul 1. În cazul 3 se execută două schimbări de culoare şi o rotaţie la dreapta, care conservă proprietatea 4, şi apoi, deoarece nu mai avem două noduri consecutive colorate cu roşu, am terminat. Corpul ciclului **cât timp** nu se va mai executa încă o dată deoarece acum $p[x]$ este colorat cu negru.

Care este timpul de execuţie al algoritmului RN-Inserează? Deoarece înălţimea unui arbore roşu-negru având n noduri este $O(\lg n)$, apelul lui Arbore-Inserează va consuma un timp $O(\lg n)$. Corpul ciclului **cât timp** se repetă numai când se execută cazul 1, şi în acest caz pointerul x urcă în arbore. Prin urmare corpul ciclului **cât timp** se poate repeta tot de $O(\lg n)$ ori. În concluzie, algoritmul RN-Inserează se execută într-un timp total de $O(\lg n)$. Este interesant de ştiut că el nu execută niciodată mai mult de două rotaţii, deoarece ciclul **cât timp** se termină în oricare din cazurile 2 şi 3.

Exerciţii

14.3-1 În linia 2 a algoritmului RN-Inserează se setează la roşu culoarea noului nod inserat x. Să observăm că dacă s-ar seta la negru culoarea lui x, atunci proprietatea 3 a arborilor roşu-negru nu ar fi încălcată. De ce nu s-a setat la negru culoarea lui x?

14.3-2 În linia 20 a algoritmului RN-Inserează se setează la negru culoarea rădăcinii. Care este avantajul acestei operaţii?

14.3-3 Desenaţi arborii roşu-negru care rezultă după inserarea succesivă a cheilor 41, 38, 31, 12, 19, 8 într-un arbore roşu-negru iniţial vid.

14.3-4 Presupunem că înălţimea neagră a fiecăruia dintre subarborii $\alpha, \beta, \gamma, \delta$ şi ε din figurile 14.5 şi 14.6 este k. Etichetaţi fiecare nod din cele două figuri cu înălţimea neagră a sa pentru a verifica respectarea proprietăţii 4 prin transformările indicate.

14.3-5 Considerăm un arbore roşu-negru format prin inserarea a n noduri, folosind algoritmul RN-Inserează. Demonstraţi că dacă $n > 1$ atunci arborele are cel puţin un nod colorat cu roşu.

14.3-6 Sugeraţi o implementare eficientă a algoritmului RN-Inserează în cazul în care reprezentarea arborilor roşu-negru nu conţine informaţii despre pointerii părinte.

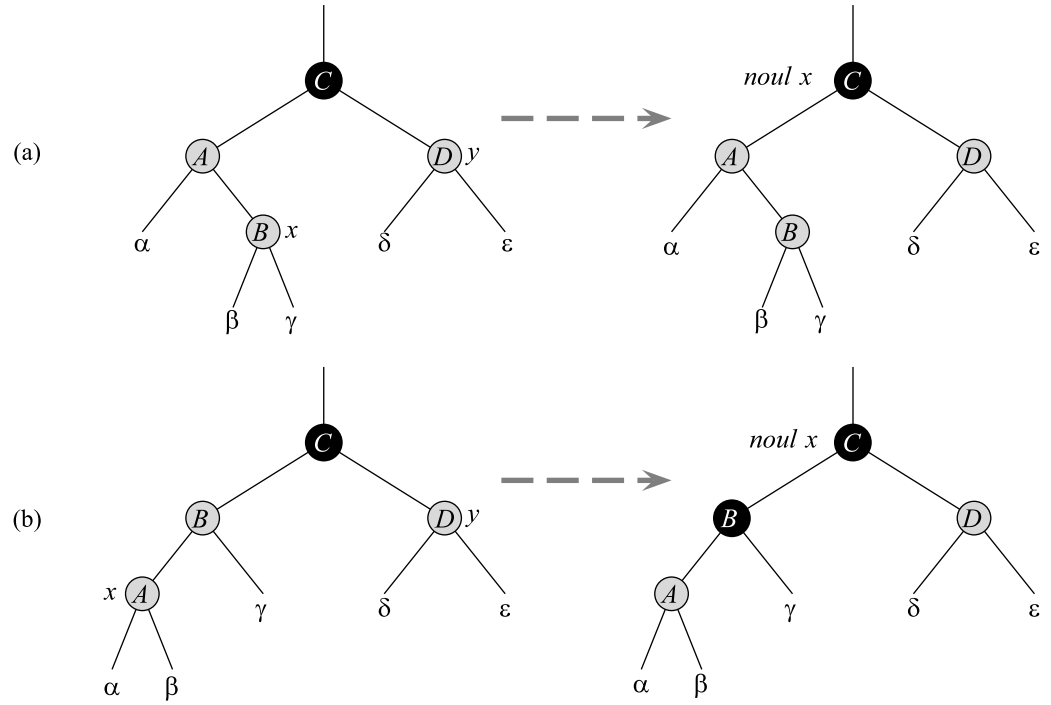


Figura 14.5 Cazul 1 al procedurii RN-Inserează. Proprietatea 3 este încălcată deoarece x şi părintele său $p[x]$ sunt ambii coloraţi cu roşu. Aceeaşi acţiune este efectuată indiferent dacă **(a)** x este fiul drept sau **(b)** x este fiul stâng. Fiecare dintre subarborii $\alpha, \beta, \gamma, \delta$ şi ε au rădăcina colorată cu negru şi toţi au aceeaşi înălţime neagră. Codul pentru cazul 1 schimbă culorile unor noduri, conservând proprietatea 4: toate drumurile în jos de la un nod fixat la frunze au acelaşi număr de noduri colorate cu negru. Ciclul **cât timp** continuă cu bunicul $p[p[x]]$ al lui x pe post de noul nod x inserat. Orice încălcare a proprietăţii 3 poate să apară acum numai între noul x, care este colorat cu roşu, şi părintele său, dacă şi acesta este colorat cu roşu.

14.4. Ştergerea

La fel ca şi celelalte operaţii de bază pe un arbore roşu-negru cu n noduri, ştergerea unui nod consumă un timp $O(\lg n)$. Ştergerea unui nod dintr-un arbore roşu-negru nu este cu mult mai complicată decât inserarea unui nod într-un astfel de arbore.

Pentru a face mai simple condiţiile la limită (extreme) din cod, vom folosi o santinelă pentru a reprezenta NIL (vezi pagina 176. Pentru un arbore roşu-negru T, santinela $nil[T]$ este un obiect cu aceleaşi câmpuri ca şi un nod obişnuit din arbore. Câmpul *culoare* al său este NEGRU iar celelalte câmpuri – p, *stânga*, *dreapta* şi *cheie* – se pot seta la valori arbitrare. În arborele roşu-negru, toţi pointerii la NIL vor fi înlocuiţi cu pointeri la santinela $nil[T]$.

Santinelele se folosesc pentru a trata unitar toate nodurile arborelui: un fiu NIL al unui nod x se va putea considera ca şi nod obişnuit al cărui părinte este x. Am putea adăuga câte un

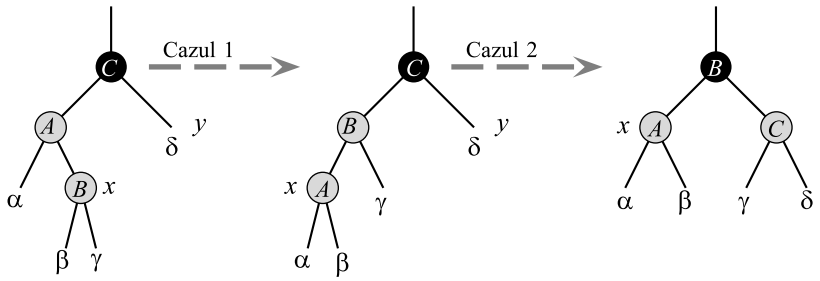


Figura 14.6 Cazurile 2 şi 3 ale procedurii RN-Inserează. Ca şi în cazul 1, proprietatea 3 este încălcată în oricare dintre cazurile 2 şi 3 deoarece x şi părintele său $p[x]$ sunt ambii coloraţi cu roşu. Fiecare dintre subarborii α, β, γ şi δ au rădăcina colorată cu negru şi toţi au aceeaşi înălţime neagră. Cazul 2 se transformă în cazul 3 printr-o rotaţie la stânga, care conservă proprietatea 4: toate drumurile în jos de la un nod fixat la frunze au acelaşi număr de noduri colorate cu negru. Cazul 3 schimbă culorile unor noduri şi efectuează o rotaţie la dreapta, care şi ea conservă proprietatea 4. Apoi execuţia ciclului **cât timp** se termină, deoarece proprietatea 3 este verificată: nu mai există două noduri consecutive colorate cu roşu.

nod santinelă distinct pentru fiecare nod NIL din arbore, astfel că părintele fiecărui nod NIL ar fi bine definit, însă această strategie ar însemna un consum suplimentar de spaţiu. În consecinţă, vom folosi o singură santinelă $nil[T]$ pentru a reprezenta toate nodurile NIL. Când vom dori să manipulăm un fiu al unui nod x, va trebui să nu uităm să setăm în prealabil $p[nil[T]]$ la x.

RN-Şterge(T, z)
1: **dacă** *stânga*$[z] = nil[T]$ sau $dreapta[z] = nil[T]$ **atunci**
2: $y \leftarrow z$
3: **altfel**
4: $y \leftarrow$ Arbore-Succesor(z)
5: **dacă** *stânga*$[y] \neq nil[T]$ **atunci**
6: $x \leftarrow$*stânga*$[y]$
7: **altfel**
8: $x \leftarrow dreapta[y]$
9: $p[x] \leftarrow p[y]$
10: **dacă** $p[y] = nil[T]$ **atunci**
11: *rădăcină*$[T] \leftarrow x$
12: **altfel dacă** $y =$*stânga*$[p[y]]$ **atunci**
13: *stânga*$[p[y]] \leftarrow x$
14: **altfel**
15: $dreapta[p[y]] \leftarrow x$
16: **dacă** $y \neq z$ **atunci**
17: $cheie[z] \leftarrow cheie[y]$
18: ▷copiază şi celelalte câmpuri ale lui y
19: **dacă** $culoare[y] =$ NEGRU **atunci**
20: RN-Şterge-Repară(T, x)
21: **returnează** y

Procedura RN-ŞTERGE reprezintă o modificare minoră a procedurii ARBORE-ŞTERGE (secţiunea 13.3). După eliminarea nodului din arbore, ea apelează o procedură ajutătoare RN-ŞTERGE-REPARĂ care schimbă culorile şi efectuează rotaţiile necesare conservării proprietăţilor roşu-negru.

Există trei deosebiri între procedurile ARBORE-ŞTERGE şi RN-ŞTERGE. În primul rând, toate referirile la NIL din ARBORE-ŞTERGE au fost înlocuite în RN-ŞTERGE prin referinţe la $nil[T]$. În al doilea rând, testul pentru $x \neq$ NIL din linia 9 a lui ARBORE-ŞTERGE a fost eliminat, iar atribuirea $p[x] \leftarrow p[y]$ se face necondiţionat în linia 9 a algoritmului RN-ŞTERGE. Astfel, dacă x este santinela $nil[T]$, pointerul său părinte va referi părintele nodului y eliminat din arbore. Ultima deosebire constă în apelul procedurii RN-ŞTERGE-REPARĂ care se execută în liniile 19–20 când y este colorat cu negru. Dacă y este colorat cu roşu, proprietăţile roşu-negru sunt conservate chiar dacă y este eliminat din arbore, deoarece în arbore nu s-au modificat înălţimile negre şi nu există noduri adiacente colorate cu roşu. Nodul x transmis procedurii RN-ŞTERGE-REPARĂ este nodul care a fost unicul fiu al lui y înainte ca y să fie eliminat, dacă y a avut un fiu diferit de NIL, sau santinela $nil[T]$ în cazul când y nu a avut fii. În ultimul caz atribuirea necondiţionată din linia 9 garantează că părintele lui x este acum nodul care a fost anterior părintele lui y, indiferent dacă x este un nod intern care conţine o cheie sau este santinela $nil[T]$.

Acum putem să examinăm modul în care procedura RN-ŞTERGE-REPARĂ restabileşte proprietăţile roşu-negru ale arborelui de căutare.

RN-ŞTERGE-REPARĂ(T, x)
1: **cât timp** $x \neq$ *rădăcină*$[T]$ şi $culoare[x] =$ NEGRU **execută**
2: **dacă** $x =$ *stânga*$[p[x]]$ **atunci**
3: $w \leftarrow dreapta[p[x]]$
4: **dacă** $culoare[w] =$ ROŞU **atunci**
5: $culoare[w] \leftarrow$ NEGRU ▷Cazul 1
6: $culoare[p[x]] \leftarrow$ ROŞU ▷Cazul 1
7: ROTEŞTE-STÂNGA$(T, p[x])$ ▷Cazul 1
8: $w \leftarrow dreapta[p[x]]$ ▷Cazul 1
9: **dacă** $culoare[$*stânga*$[w]] =$ NEGRU şi $culoare[dreapta[w]] =$ NEGRU **atunci**
10: $culoare[w] \leftarrow$ ROŞU ▷Cazul 2
11: $x \leftarrow p[x]$ ▷Cazul 2
12: **altfel**
13: **dacă** $culoare[dreapta[w]] =$ NEGRU **atunci**
14: $culoare[$*stânga*$[w]] \leftarrow$ NEGRU ▷Cazul 3
15: $culoare[w] \leftarrow$ ROŞU ▷Cazul 3
16: ROTEŞTE-DREAPTA(T, w) ▷Cazul 3
17: $w \leftarrow dreapta[p[x]]$ ▷Cazul 3
18: $culoare[w] \leftarrow culoare[p[x]]$ ▷Cazul 4
19: $culoare[p[x]] \leftarrow$ NEGRU ▷Cazul 4
20: $culoare[dreapta[w]] \leftarrow$ NEGRU ▷Cazul 4
21: ROTEŞTE-STÂNGA$(T, p[x])$ ▷Cazul 4
22: $x \leftarrow$ *rădăcină*$[T]$ ▷Cazul 4
23: **altfel**
24: (La fel ca în clauza **atunci** interschimbând "dreapta" cu "stânga")
25: $culoare[x] \leftarrow$ NEGRU

Dacă nodul y eliminat în RN-ŞTERGE este negru, ştergerea lui are ca efect micşorarea cu 1 a numărului de noduri negre de pe fiecare drum care a conţinut acest nod. Prin urmare, proprietatea 4 nu este respectată pentru toţi strămoşii nodului y. Acest inconvenient se poate corecta considerând că nodul x are un "negru suplimentar" sau este "dublu colorat cu negru". Cu alte cuvinte, dacă adăugăm 1 la numărul de noduri colorate cu negru de pe orice drum care îl conţine pe x, atunci, în interpretarea de mai sus, proprietatea 4 este verificată. Când vom elimina nodul y colorat cu negru, vom "împinge" culoarea sa neagră în fiul său. Singura problemă care apare acum este aceea că nodul x ar putea fi "dublu colorat cu negru", ceea ce încalcă proprietatea 1.

Procedura RN-ŞTERGE-REPARĂ încearcă să refacă proprietatea 1. Scopul ciclului **cât timp** din liniile 1–24 este de a muta în sus în arbore nodul "dublu colorat cu negru" până când

1. x referă un nod colorat cu roşu, caz în care se va colora x cu negru în linia 25
2. x referă rădăcina, caz în care nodul "dublu colorat cu negru" este doar "eliminat" sau
3. se pot efectua rotaţii şi recolorări adecvate.

În corpul ciclului **cât timp**, x referă întotdeauna un nod colorat cu negru, care nu este rădăcină şi care este "dublu colorat cu negru". Linia 2 determină dacă x este fiul stâng sau drept al părintelui său $p[x]$. (Codul este dat pentru situaţia în care x este fiul stâng al părintelui său; cazul când x este fiu drept – plasat în linia 24 – este simetric). Se foloseşte un pointer w la fratele lui x. Deoarece nodul x este "dublu colorat cu negru", nodul w nu poate fi $nil[T]$; dacă ar fi, numărul de noduri colorate cu negru pe drumul de la $p[x]$ la NIL, ar fi mai mic decât numărul de noduri colorate cu negru pe drumul de la $p[x]$ la x.

Cele patru cazuri expuse în cod sunt ilustrate în figura 14.7. Înainte de a examina detaliat fiecare caz în parte, să aruncăm o privire generală asupra modului în care se verifică respectarea proprietăţii 4 pentru transformările efectuate în toate aceste cazuri. Ideea de bază este aceea că în fiecare caz numărul de noduri colorate cu negru de la rădăcină (incluzând şi rădăcina) la fiecare dintre subarborii $\alpha, \beta, \ldots, \zeta$ este conservat prin transformarea efectuată. De exemplu, în figura 14.7(a), care corespunde cazului 1, numărul de noduri colorate cu negru de la rădăcină la oricare dintre subarborii α sau β este 3, atât înainte cât şi după transformare. (Să ne reamintim că pointerul x referă un nod "dublu colorat cu negru"). Similar, numărul de noduri negre de la rădăcină la oricare dintre subarborii $\gamma, \delta, \varepsilon$ sau ζ este 2, atât înainte, cât şi după transformare. În figura 14.7(b) numărarea trebuie să ia în considerare şi culoarea c, care poate fi roşu sau negru. Dacă notăm *număr*(ROŞU) = 0 şi *număr*(NEGRU) = 1, atunci numărul de noduri negre de la rădăcină la α este 2+*număr*(c), atât înainte, cât şi după transformare. Celelalte cazuri se pot verifica într-o manieră similară (exerciţiul 14.4-5).

Cazul 1 (liniile 5–8 din RN-ŞTERGE-REPARĂ şi figura 14.7(a)) apare când nodul w, fratele nodului x, este colorat cu roşu. Deoarece w trebuie să aibă fiii coloraţi cu negru, putem interschimba culorile lui w şi $p[x]$ şi apoi să efectuăm o rotaţie la stânga în $p[x]$ fără a încălca nici una dintre proprietăţile roşu-negru. Noul frate al lui x, unul dintre fiii lui w, este acum colorat cu negru şi prin urmare am transformat cazul 1 într-unul din cazurile 2, 3 sau 4.

Cazurile 2, 3 şi 4 apar când nodul w este colorat cu negru; ele se deosebesc prin culorile fiilor lui w. În cazul 2 (liniile 10–11 din RN-ŞTERGE-REPARĂ şi figura 14.7(b)), ambii fii ai lui w sunt coloraţi cu negru. Deoarece şi w este colorat cu negru, vom "scoate afară un negru" atât de la x cât şi de la w, lăsând x numai cu un negru, punând w pe roşu şi adăugând un negru suplimentar

la $p[x]$, care va deveni "dublu colorat cu negru" prin eliminarea negrului suplimentar din fiul său. Apoi corpul ciclului **cât timp** se poate repeta cu $p[x]$ pe post de nod x. Să observăm că dacă se ajunge la cazul 2 din cazul 1, atunci culoarea c a noului nod x este roşu, deoarece nodul $p[x]$ original a fost roşu, şi prin urmare, după execuţia cazului 2, ciclul se va termina când se testează condiţia de ciclare.

Cazul 3 (liniile 14–17 din RN-Şterge-Repară şi figura 14.7(c)) apare când w este colorat cu negru, fiul său stâng este colorat cu roşu, iar fiul său drept este colorat cu negru. Putem interschimba culorile lui w şi ale fiului său stâng *stânga*$[w]$ şi apoi să efectuăm o rotaţie la dreapta în w fără a încălca proprietăţile roşu-negru. Noul frate w al lui x este acum un nod colorat cu negru care are un fiu drept colorat cu roşu şi prin urmare am transformat cazul 3 în cazul 4.

Cazul 4 (liniile 18–22 din RN-Şterge-Repară şi figura 14.7(d)) apare când fratele w al nodului x este colorat cu negru, iar fiul drept al lui w este colorat cu roşu. Efectuând anumite schimbări de culoare şi apoi o rotaţie la stânga în $p[x]$, se poate elimina "negrul suplimentar" din x fără a încălca proprietăţile roşu-negru. Setarea lui x ca rădăcină provoacă terminarea ciclului **cât timp** când se testează condiţia de continuare a ciclării.

Care este timpul de execuţie al procedurii RN-Şterge? Deoarece înălţimea unui arbore roşu-negru având n noduri este $O(\lg n)$, costul total al procedurii fără a include şi apelul lui RN-Şterge-Repară este $O(\lg n)$. În RN-Şterge-Repară, fiecare din cazurile 1, 3 şi 4 se termină după efectuarea unui număr constant de modificări de culoare urmate de cel mult trei rotaţii. Singurul caz în care corpul ciclului **cât timp** se repetă este cazul 2, şi în acesta pointerul x se mută în sus în arbore de cel mult $O(\lg n)$ ori, fără a se efectua rotaţii. În consecinţă, RN-Şterge-Repară consumă un timp de $O(\lg n)$ şi efectuează cel mult trei rotaţii, iar timpul total de execuţie pentru RN-Şterge este de $O(\lg n)$.

Exerciţii

14.4-1 Arătaţi că dacă un arbore roşu-negru are rădăcina colorată cu negru înainte de apelul procedurii RN-Şterge, atunci culoarea rădăcinii rămâne aceeaşi şi după apel.

14.4-2 În exerciţiul 14.3-3 aţi determinat arborele roşu-negru care rezultă prin inserarea succesivă a cheilor 41, 38, 31, 12, 19, 8 într-un arbore iniţial vid. Desenaţi arborii roşu-negru care rezultă prin ştergerea succesivă a cheilor în ordinea 8, 12, 19, 31, 38, 41.

14.4-3 În care linii de cod din procedura RN-Şterge-Repară se inspectează sau se modifică santinela $nil[T]$?

14.4-4 Simplificaţi codul procedurii Roteşte-Stânga prin folosirea unei santinele pentru NIL şi a unei alte santinele pentru a memora pointerul la rădăcină.

14.4-5 Calculaţi numărul de noduri negre de la rădăcină la fiecare dintre subarborii $\alpha, \beta, \ldots, \zeta$ pentru fiecare din cazurile ilustrate în figura 14.7 şi verificaţi dacă fiecare număr rămâne acelaşi după transformare. Dacă un nod are culoarea c sau c', folosiţi notaţia *număr*(c) sau *număr*(c') pentru a exprima respectivul număr.

14.4-6 Presupunem că se inserează un nod x într-un arbore roşu-negru folosind procedura RN-Inserează şi apoi acesta este şters imediat cu procedura RN-Şterge. Întrebarea este: arborele care rezultă după ştergerea lui x este identic cu arborele de dinainte de inserarea lui x? Justificaţi răspunsul.

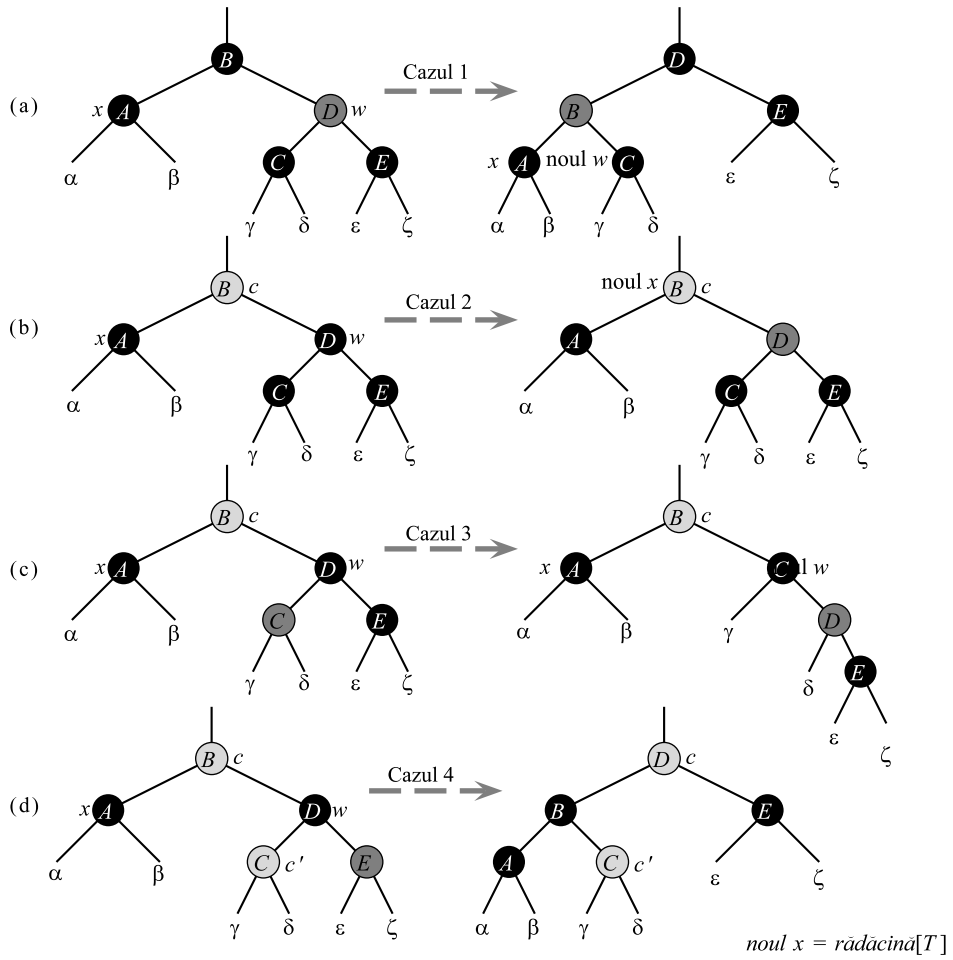


Figura 14.7 Cazurile conţinute în corpul ciclului **cât timp** al procedurii RN-Şterge. Nodurile înnegrite sunt colorate cu negru, cele haşurate cu gri închis sunt colorate cu roşu, iar cele haşurate cu gri deschis, care pot să fie colorate fie cu roşu, fie cu negru, sunt notate cu c şi c'. Literele $\alpha, \beta, \ldots, \zeta$ reprezintă subarbori arbitrari. În fiecare caz, configuraţia din stânga se transformă în configuraţia din dreapta prin schimbarea unor culori şi/sau efectuarea unei rotaţii. Nodul referit de x "este dublu colorat cu negru". Singurul caz care provoacă repetarea ciclării este cazul 2. **(a)** Cazul 1 se transformă într-unul din cazurile 2, 3 sau 4 prin interschimbarea culorilor nodurilor B şi D şi efectuarea unei rotaţii la stânga. **(b)** În cazul 2, negrul suplimentar reprezentat de pointerul x se mută în sus în arbore prin colorarea nodului D cu roşu şi setarea lui x pentru a referi nodul B. Dacă se ajunge la cazul 2 de la cazul 1, ciclul **cât timp** se termină deoarece culoarea c este roşu. **(c)** Cazul 3 se transformă în cazul 4 prin interschimbarea culorilor nodurilor C şi D şi efectuarea unei rotaţii la dreapta. **(d)** În cazul 4, negrul suplimentar reprezentat de x se poate elimina prin schimbarea unor culori şi efectuarea unei rotaţii la stânga (fără a încălca proprietăţile roşu-negru) şi apoi ciclul se termină.

Probleme

14-1 Mulţimi dinamice persistente

În timpul descrierii unui algoritm avem câteodată nevoie să păstrăm versiunile anterioare ale unei mulţimi dinamice când aceasta se modifică. O astfel de mulţime se numeşte ***persistentă***. O modalitate de implementare a unei mulţimi persistente este copierea întregii mulţimi ori de câte ori aceasta se modifică, însă această abordare poate încetini programul şi în acelaşi timp consumă mult spaţiu. În unele situaţii putem să procedăm mai eficient.

Fie o mulţime persistentă S cu operaţiile INSEREAZĂ, ŞTERGE şi CAUTĂ, care este implementată folosind arbori binari de căutare, după cum se arată în figura 14.8(a). Pentru fiecare versiune a mulţimii se memorează o rădăcină distinctă. Pentru a insera cheia 5 în mulţime, se creează un nod nou având cheia 5. Acest nod devine fiul stâng al unui nod nou cu cheia 7, deoarece nu putem modifica nodul existent având cheia 7. Similar, noul nod având cheia 7 devine fiul stâng al unui nod nou cu cheia 8 al cărui fiu din dreapta este un nod existent având cheia 10. Noul nod cu cheia 8 devine, la rândul său, fiul drept al unei rădăcini noi r' având cheia 4 al cărei fiu stâng este nodul existent cu cheia 3. Prin urmare se va copia numai o parte din arborele iniţial, aşa cum se ilustrează în figura 14.8(b).

Presupunem că fiecare nod din arbore are câmpurile *cheie*, *stânga* şi *dreapta*, însă nu are un câmp părinte. (Vezi şi exerciţiul 14.3-6).

a. Identificaţi nodurile care trebuie modificate la inserarea unei chei k şi la ştergerea unui nod y într-un/dintr-un arbore binar de căutare general şi persistent.

b. Scrieţi o procedură PERSISTENT-ARBORE-INSEREAZĂ care, pentru un arbore persistent T şi o cheie k de inserat, întoarce un arbore persistent nou T' care este rezultatul inserării cheii k în T.

c. Dacă înălţimea unui arbore binar de căutare persistent T este h, care sunt cerinţele de timp şi spaţiu pentru implementarea pe care aţi dat-o pentru PERSISTENT-ARBORE-INSEREAZĂ? (Cerinţele de spaţiu sunt proporţionale cu numărul de noduri noi alocate).

d. Presupunem acum că fiecare nod are şi un câmp părinte. În acest caz, PERSISTENT-ARBORE-INSEREAZĂ are nevoie de copierea unor informaţii suplimentare. Demonstraţi că PERSISTENT-ARBORE-INSEREAZĂ va avea nevoie de un timp şi spaţiu $\Omega(n)$, unde n este numărul de noduri din arbore.

e. Arătaţi cum se pot folosi arborii roşu-negru pentru a se garanta că cerinţele de timp şi spaţiu în cazul cel mai defavorabil sunt de $O(\lg n)$ pentru inserare sau ştergere.

14-2 Operaţii de uniune pe arbori roşu-negru

Operaţia de ***uniune*** are ca argumente două mulţimi dinamice S_1 şi S_2 şi un element x astfel încât pentru orice $x_1 \in S_1$ şi $x_2 \in S_2$ are loc relaţia $cheie[x_1] \leq cheie[x] \leq cheie[x_2]$. Ea returnează o mulţime $S = S_1 \cup \{x\} \cup S_2$. În această problemă investigăm modul de implementare al operaţiei de uniune pe arbori roşu-negru.

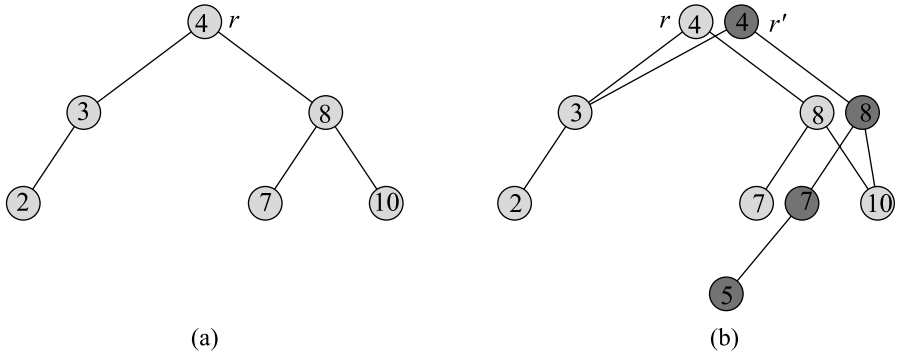


Figura 14.8 **(a)** Un arbore binar de căutare având cheile 2, 3, 4, 7, 8, 10. **(b)** Arborele binar de căutare persistent care rezultă prin inserarea cheii 5. Cea mai recentă versiune a mulţimii constă din nodurile ce se pot accesa din rădăcina r', iar versiunea anterioară este formată din nodurile ce se pot accesa din rădăcina r. Nodurile haşurate cu gri închis sunt cele adăugate pentru inserarea cheii 5.

a. Fiind dat un arbore roşu-negru T, vom memora înălţimea neagră a sa în câmpul $bh[T]$. Arătaţi că acest câmp se poate întreţine prin algoritmii RN-INSEREAZĂ şi RN-ŞTERGE fără a fi nevoie de spaţiu suplimentar de memorie în arbore şi fără a mări timpii de execuţie asimptotici. Arătaţi că înălţimea neagră a fiecărui nod se poate determina pe parcursul coborârii în T într-un timp $O(1)$ pentru fiecare nod vizitat.

Dorim să implementăm operaţia RN-UNIUNE(T_1, x, T_2) care distruge T_1 şi T_2 şi întoarce arborele roşu-negru $T = T_1 \cup \{x\} \cup T_2$. Notăm cu n numărul total de noduri din T_1 şi T_2.

b. Fără a restrânge generalitatea, presupunem că $bh[T_1] \geq bh[T_2]$. Descrieţi un algoritm de timp $O(\lg n)$ care determină nodul colorat cu negru din T_1 cu cea mai mare cheie şi care are înălţimea neagră egală cu $bh[T_2]$.

c. Notăm cu T_y subarborele având rădăcina y. Descrieţi cum se poate înlocui T_y cu $T_y \cup \{x\} \cup T_2$ într-un timp $O(1)$ şi fără a distruge proprietatea arborelui binar de căutare.

d. Ce culoare trebuie atribuită lui x pentru a conserva proprietăţile roşu-negru 1, 2 şi 4? Descrieţi cum se poate reface proprietatea 3 într-un timp $O(\lg n)$.

e. Arătaţi că timpul de execuţie al procedurii RN-UNIUNE este $O(\lg n)$.

Note bibliografice

Ideea echilibrării arborelui de căutare se datorează lui Adel'son-Vel'skii şi Landis [2], care au introdus o clasă de arbori de căutare echilibraţi numiţi "arbori AVL". În arborii AVL, echilibrarea este menţinută prin rotaţii, însă pentru a reface echilibrarea ar putea fi necesare $\Theta(\lg n)$ rotaţii după ştergerea unui nod dintr-un arbore având n noduri. O altă clasă de arbori de căutare, numiţi "2-3 arbori", a fost introdusă de J. E. Hopcroft în 1970 (manuscris nepublicat). Într-un 2-3 arbore echilibrarea se menţine prin manipularea gradelor nodurilor din arbore. Capitolul 19

se ocupă cu B-arborii, care sunt o generalizare a 2-3 arborilor şi au fost introduşi de Bayer şi McCreight [18].

Arborii roşu-negru au fost inventaţi de Bayer [17] sub numele de "B-arbori binari simetrici". Proprietăţile lor au fost studiate în detaliu de Guibas şi Sedgewick [93], care au introdus şi convenţia de colorare roşu/negru.

Dintre celelalte clase de arbori binari echilibraţi, probabil cea mai interesantă este cea a "arborilor splay", introduşi de Sleator şi Tarjan [177], care sunt "auto-ajustabili". (O bună descriere a arborilor splay este dată de Tarjan [188]). Arborii splay întreţin echilibrarea fără nici o condiţie explicită de echilibrare, cum ar fi de exemplu culoarea. În locul unei astfel de condiţii, se execută "operaţii splay" în arbore (care implică rotaţii) ori de câte ori se face o accesare. Costul amortizat (vezi capitolul 18) al fiecărei operaţii pe un arbore cu n noduri este $O(\lg n)$.

15 Îmbogăţirea structurilor de date

În ingineria programării apar uneori situaţii care nu necesită decât o structură de date "clasică, exact ca în carte" – ca de exemplu lista dublu înlănţuită, tabela de dispersie sau arborele binar de căutare – însă majoritatea problemelor care trebuie rezolvate impun o anumită doză de creativitate. Totuşi, doar în puţine cazuri este nevoie să se creeze un tip complet nou de structură de date. Mult mai frecventă este situaţia când este suficientă adăugarea de informaţii (câmpuri) suplimentare la o structură de date clasică pentru ca aceasta să se poată folosi în aplicaţia dorită. Desigur, îmbogăţirea structurilor de date nu este întotdeauna simplă, deoarece informaţia adăugată trebuie să fie actualizată şi întreţinută prin operaţiile ordinare ale structurii de date respective.

Acest capitol prezintă două structuri de date care se construiesc prin îmbogăţirea arborilor roşu-negru. Secţiunea 15.1 descrie o structură de date care posedă operaţiile generale de statistică de ordine pe o mulţime dinamică. Cu ajutorul acestora se poate determina rapid al i-lea cel mai mic număr din mulţime sau rangul unui element dat în ordonarea totală a mulţimii. Secţiunea 15.2 abstractizează procesul de îmbogăţire a unei structuri de date şi dă o teoremă care poate simplifica îmbogăţirea arborilor roşu-negru. Secţiunea 15.3 foloseşte această teoremă pentru a uşura proiectarea unei structuri de date pentru întreţinerea unei mulţimi dinamice de intervale, ca de exemplu intervalele de timp. Fiind dat un interval de interogare, se poate determina rapid un interval din mulţime care se suprapune cu intervalul dat.

15.1. Statistici dinamice de ordine

Capitolul 10 a introdus noţiunea de statistică de ordine. Specific, a i-a statistică de ordine a unei mulţimi de n elemente, unde $i \in \{1, 2, \ldots, n\}$, este elementul din mulţime care are a i-a cea mai mică cheie. Am arătat că orice statistică de ordine se poate determina într-un timp $O(n)$ în cazul unei mulţimi neordonate. În această secţiune vom vedea cum se pot modifica arborii roşu-negru pentru ca orice statistică de ordine să se determine într-un timp $O(\lg n)$ şi vom afla cum se determină ***rangul*** unui element – poziţia acestuia în ordonarea liniară a mulţimii – tot într-un timp $O(\lg n)$.

Figura 15.1 prezintă o structură de date care posedă operaţii rapide de statistici de ordine. Un ***arbore de statistică de ordine*** T este de fapt un arbore roşu-negru care conţine o informaţie suplimentară în fiecare nod. La câmpurile uzuale pentru un nod x dintr-un arbore roşu-negru $cheie[x]$, $culoare[x]$, $p[x]$, $\mathsf{stânga}[x]$ şi $dreapta[x]$ se adaugă un câmp nou, $dimensiune[x]$. Acest câmp conţine numărul de noduri (interne) din subarborele cu rădăcina x (incluzându-l şi pe x), adică dimensiunea subarborelui. Dacă facem convenţia $dimensiune[\text{NIL}] = 0$, atunci are loc identitatea:

$$dimensiune[x] = dimensiune[\mathsf{stânga}[x]] + dimensiune[dreapta[x]] + 1$$

(Pentru a gestiona adecvat condiţia extremă pentru NIL, o implementare efectivă va face explicit testul $x \neq \text{NIL}$ anterior accesării câmpului $dimensiune$ sau, mai simplu, ca în secţiunea 14.4, va folosi o santinelă $nil[T]$ pentru a-l reprezenta pe NIL, cu $dimensiune[nil[T]] = 0$.)

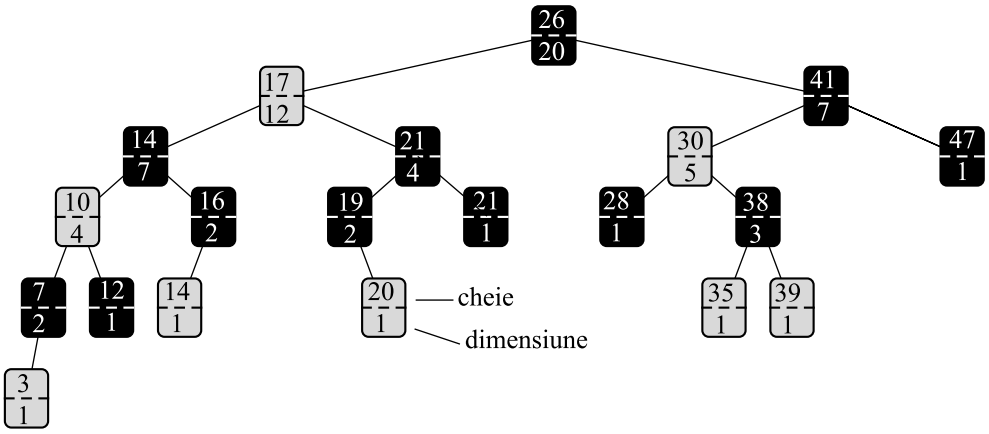


Figura 15.1 Un arbore de statistică de ordine care este de fapt un arbore roşu-negru îmbogăţit. Nodurile haşurate cu gri sunt roşii, iar nodurile haşurate cu negru sunt negre. Pe lângă câmpurile uzuale, fiecare nod x are un câmp $dimensiune[x]$ care reprezintă numărul de noduri din subarborele având rădăcina x.

Regăsirea unui element cu rangul cunoscut

Înainte de a arăta cum se întreţine informaţia de dimensiune în timpul inserării şi ştergerii, să examinăm implementarea a două interogări de statistică de ordine care folosesc această informaţie suplimentară. Începem cu o operaţie care regăseşte un element cu rangul cunoscut. Procedura SO-SELECTEAZĂ(x, i) returnrază un pointer la nodul care conţine a i-a cea mai mică cheie din subarborele având rădăcina x. Pentru a determina a i-a cea mai mică cheie dintr-un arbore de statistică de ordine T vom face apelul SO-SELECTEAZĂ($\mathit{rădăcină}[T], i$).

SO-SELECTEAZĂ(x, i)
1: $r \leftarrow dimensiune[\mathit{stânga}[x]] + 1$
2: **dacă** $i = r$ **atunci**
3: **returnează** x
4: **altfel dacă** $i < r$ **atunci**
5: **returnează** SO-SELECTEAZĂ($\mathit{stânga}[x], i$)
6: **altfel**
7: **returnează** SO-SELECTEAZĂ($dreapta[x], i - r$)

Ideea pe care se bazează SO-SELECTEAZĂ este similară celei de la algoritmii de selecţie prezentaţi în capitolul 10. Valoarea lui $dimensiune[\mathit{stânga}[x]]$ reprezintă numărul de noduri care sunt inspectate anterior lui x, la traversarea în inordine a subarborelui având rădăcina x. Prin urmare, $dimensiune[\mathit{stânga}[x]] + 1$ este rangul lui x în subarborele având rădăcina x.

În linia 1 a algoritmului SO-SELECTEAZĂ se calculează r, rangul nodului x în subarborele având rădăcina x. Dacă $i = r$, atunci nodul x este al i-lea cel mai mic element şi x va fi returnat în linia 3. Dacă $i < r$, atunci al i-lea cel mai mic element se va găsi în subarborele stâng al lui x, prin urmare se va apela recursiv algoritmul pentru subarborele având rădăcina $\mathit{stânga}[x]$ în linia 5. Dacă $i > r$, atunci al i-lea cel mai mic element se va găsi în subarborele drept al lui x. Deoarece în subarborele având rădăcina x sunt r elemente care se inspectează înaintea

subarborelui drept al lui x la traversarea în inordine a arborelui, al i-lea cel mai mic element din subarborele având rădăcina x este al $(i - r)$-lea cel mai mic element din subarborele având rădăcina $dreapta[x]$. Acest element se determină recursiv în linia 7.

Pentru a vedea cum funcţionează SO-SELECTEAZĂ, să considerăm căutarea pentru al 17-lea cel mai mic element din arborele de statistică de ordine din figura 15.1. Se începe cu x egal cu rădăcina, a cărei cheie este 26, şi cu $i = 17$. Deoarece dimensiunea subarborelui stâng al lui 26 este 12, rangul rădăcinii 26 va fi 13. Prin urmare, ştim deja că nodul cu rangul 17 este al 17 - 13 = 4-lea cel mai mic element din subarborele drept al lui 26. După apelul recursiv, x este nodul având cheia 41, iar $i = 4$. Deoarece dimensiunea subarborelui stâng al lui 41 este 5, rangul său în subarborele în care acesta este rădăcină, este 6. Prin urmare ştim deja că nodul cu rangul 4 este în al 4-lea cel mai mic element al subarborelui stâng al lui 41. După apelul recursiv, x este nodul având cheia 30, iar rangul său în subarborele pentru care este rădăcină este 2. Prin urmare se face un nou apel recursiv pentru a determina al 4 - 2 = 2-lea cel mai mic element din subarborele având rădăcina 38. La acest apel vom constata că subarborele stâng are dimensiunea 1, ceea ce înseamnă că rădăcina 38 este al 2-lea cel mai mic element. În consecinţă, acest ultim apel va întoarce un pointer la nodul având cheia 38 şi execuţia procedurii se termină.

Deoarece fiecare apel recursiv coboară un nivel în arborele de statistică de ordine, timpul total pentru SO-SELECTEAZĂ este în cel mai defavorabil caz proporţional cu înălţimea arborelui. Deoarece arborele este un arbore roşu-negru, înălţimea sa este $O(\lg n)$, unde n este numărul de noduri. Prin urmare, timpul de execuţie pentru SO-SELECTEAZĂ este $O(\lg n)$ pentru o mulţime dinamică având n elemente.

Determinarea rangului unui element

Fiind dat un pointer la un nod x dintr-un arbore de statistică de ordine T, procedura SO-RANG returnnează poziţia lui x în ordinea liniară dată de traversarea în inordine a lui T.

SO-RANG(T, x)
1: $r \leftarrow dimensiune[\textit{stânga}[x]] + 1$
2: $y \leftarrow x$
3: **cât timp** $y \neq \textit{rădăcină}[T]$ **execută**
4: **dacă** $y = dreapta[p[y]]$ **atunci**
5: $r \leftarrow r + dimensiune[\textit{stânga}[p[y]]] + 1$
6: $y \leftarrow p[y]$
7: **returnează** r

Procedura funcţionează după cum urmează. Rangul lui x poate fi considerat a fi numărul de noduri inspectate înaintea lui x la traversarea în inordine a arborelui plus 1. Se foloseşte următorul invariant: la începutul corpului ciclului **cât timp** din liniile 3–6, r este rangul lui $cheie[x]$ din subarborele având rădăcina y. Acest invariant se calculează după cum este descris în continuare. În linia 1 se setează r la rangul lui $cheie[x]$ din subarborele având rădăcina x. Datorită atribuirii $y \leftarrow x$ din linia 2 invariantul este adevărat pentru prima execuţie a testului din linia 3. La fiecare iteraţie a corpului ciclului **cât timp**, se lucrează cu subarborele care are rădăcina $p[y]$. A fost contorizat deja numărul de noduri din subarborele având rădăcina y care precede pe x la traversarea în inordine, aşa că va trebui să adunăm la r numărul de noduri din subarborele având rădăcina fratele lui y care-l precede pe x la traversarea în inordine, plus 1, pentru $p[y]$ dacă şi $p[y]$ îl precede pe x. Dacă y este un fiu stâng, atunci atât $p[y]$, cât şi toate

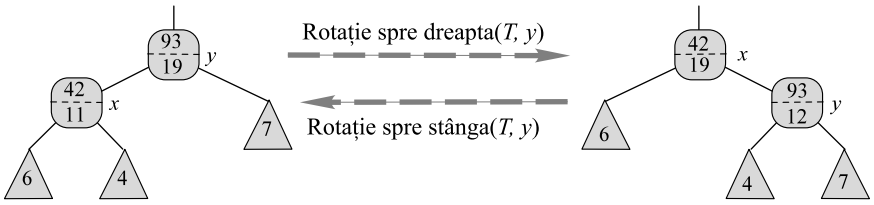


Figura 15.2 Actualizarea dimensiunii subarborilor în timpul rotaţiilor. Cele două câmpuri *dimensiune* care trebuie actualizate sunt cele care sunt la capetele legăturii în jurul căreia se efectuează rotaţia. Actualizările sunt locale, necesitând numai informaţia *dimensiune* memorată în x, y şi în rădăcinile subarborilor ilustraţi prin triunghiuri.

nodurile din subarborele drept al lui $p[y]$ sunt inspectate după x la traversarea în inordine, deci r nu se modifică. Altfel, y este un fiu drept şi toate nodurile din subarborele stâng al lui $p[y]$ sunt inspectate înaintea lui x, ca şi $p[y]$. Prin urmare, în linia 5 se va aduna la valoarea curentă a lui r numărul $dimensiune[\textit{stânga}[p[y]]] + 1$. Datorită atribuirii $y \leftarrow p[y]$ invariantul este adevărat pentru următoarea iteraţie. Când $y =$ *rădăcină*$[T]$, procedura va returna valoarea lui r, care este acum rangul lui $cheie[x]$.

De exemplu, execuţia algoritmului SO-RANG pe arborele de statistică de ordine din figura 15.1 pentru a determina rangul nodului având cheia 38 va produce următoarea secvenţă de valori pentru $cheie[y]$ şi r înregistrate la începutul corpului ciclului **cât timp**:

iteraţia	$cheie[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17

Procedura va returna rangul 17.

Deoarece fiecare iteraţie a ciclului **cât timp** consumă un timp de $O(1)$, iar y urcă un nivel în arbore la fiecare iteraţie, timpul de execuţie al algoritmului SO-RANG este, în cel mai defavorabil caz, proporţional cu înălţimea arborelui: $O(\lg n)$ pentru un arbore de statistică de ordine având n noduri.

Întreţinerea dimensiunilor subarborilor

Fiind dată valoarea câmpului *dimensiune* din fiecare nod, procedurile SO-SELECTEAZĂ şi SO-RANG pot calcula rapid informaţia de statistică de ordine. Dacă însă aceste câmpuri nu se pot întreţine eficient prin operaţiile de bază ce modifică arborii roşu-negru, strădania noastră de până acum a fost în zadar. Vom arăta acum că dimensiunile subarborilor se pot întreţine atât pentru inserare cât şi pentru ştergere, fără afectarea timpilor asimptotici de execuţie ai acestor operaţii.

Am remarcat în secţiunea 14.3 că operaţia de inserare a unui nod într-un arbore roşu-negru constă din două faze. În prima fază se coboară în arbore începând de la rădăcină şi se inserează nodul nou ca fiu al unui nod existent. A doua fază înseamnă urcarea în arbore, modificând culorile şi efectuând la sfârşit rotaţii pentru a conserva proprietăţile roşu-negru.

Pentru a întreţine dimensiunile subarborilor în prima fază, se va incrementa câmpul $dimensiune[x]$ pentru fiecare nod x de pe drumul traversat de la rădăcină în jos spre frunze. Noul nod care se adaugă va primi valoarea 1 pentru câmpul său *dimensiune*. Deoarece pe drumul parcurs sunt $O(\lg n)$ noduri, costul suplimentar pentru întreţinerea câmpurilor *dimensiune* va fi şi el $O(\lg n)$.

În faza a doua, singurele modificări structurale ale arborelui roşu-negru sunt produse de rotaţii, care sunt cel mult două. Mai mult, rotaţia este o operaţie locală: ea invalidează numai câmpurile *dimensiune* din nodurile incidente (de la capetele) legăturii în jurul căreia se efectuează rotaţia. La codul procedurii ROTEŞTE-STÂNGA(T, x) dat în secţiunea 14.2, vom adăuga următoarele două linii:

1: $dimensiune[y] \leftarrow dimensiune[x]$
2: $dimensiune[x] \leftarrow dimensiune[\mathsf{stânga}[x]] + dimensiune[dreapta[x]] + 1$

Figura 15.2 ilustrează modul în care sunt actualizate câmpurile *dimensiune*. Pentru procedura ROTEŞTE-DREAPTA modificările sunt simetrice.

Deoarece la inserarea unui nod într-un arbore roşu-negru sunt necesare cel mult două rotaţii, se consumă suplimentar doar un timp $O(1)$ pentru actualizarea câmpurilor *dimensiune* în faza a doua a inserării. În concluzie, timpul total necesar pentru inserarea unui nod într-un arbore de statistică de ordine având n noduri este $O(\lg n)$ – acelaşi din punct de vedere asimptotic cu cel pentru arborii roşu-negru obişnuiţi.

Ştergerea unui nod dintr-un arbore roşu-negru are de asemenea două faze: prima operează pe arborele binar de căutare, iar a doua necesită cel mult trei rotaţii şi în rest nu efectuează alte modificări structurale ale arborelui. (Vezi secţiunea 14.4.) Prima fază elimină din arbore un nod y. Pentru a actualiza dimensiunile subarborilor, vom traversa un drum în sus de la nodul y la rădăcină şi vom decrementa câmpurile *dimensiune* ale nodurilor de pe acest drum. Deoarece drumul are lungimea $O(\lg n)$ în cazul unui arbore roşu-negru având n noduri, timpul suplimentar consumat cu întreţinerea câmpurilor *dimensiune* în prima fază a ştergerii unui nod este $O(\lg n)$. Cele $O(1)$ rotaţii din faza a doua a ştergerii sunt gestionate în aceeaşi manieră ca şi în cazul inserării. Prin urmare, atât inserarea cât şi ştergerea, inclusiv întreţinerea câmpurilor *dimensiune*, consumă un timp $O(\lg n)$ pentru un arbore de statistică de ordine având n noduri.

Exerciţii

15.1-1 Arătaţi cum operează SO-SELECTEAZĂ$(T, 10)$ pe arborele roşu-negru T din figura 15.1.

15.1-2 Arătaţi cum operează SO-RANG(T, x) pe arborele roşu-negru T din figura 15.1 şi nodul x cu $cheie[x] = 35$.

15.1-3 Scrieţi o versiune nerecursivă a lui SO-SELECTEAZĂ.

15.1-4 Scrieţi o procedură recursivă SO-RANG-CHEIE(T, k) care are ca date de intrare un arbore de statistică de ordine T şi o cheie k şi întoarce rangul cheii k în mulţimea dinamică reprezentată prin T. Presupuneţi că valorile cheilor din T sunt distincte.

15.1-5 Fiind dat un element x dintr-un arbore de statistică de ordine având n noduri şi un număr natural i, cum se poate determina într-un timp $O(\lg n)$ al i-lea succesor al lui x în ordinea liniară a arborelui?

15.1-6 Să observăm că toate referirile la câmpul *dimensiune* din procedurile SO-SELECTEAZĂ şi SO-RANG au ca scop calcularea rangului lui x în subarborele având rădăcina x. În consecinţă, presupunem că memorăm în fiecare nod rangul său în subarborele pentru care el este rădăcină. Arătaţi cum se poate întreţine această informaţie la inserare şi ştergere. (Reamintim că aceste operaţii au nevoie de rotaţii.)

15.1-7 Arătaţi cum se poate folosi un arbore de statistică de ordine pentru a număra inversiunile (vezi problema 1-3) dintr-un tablou de dimensiune n într-un timp $O(n \lg n)$.

15.1-8 $\star$ Considerăm n coarde într-un cerc, fiecare definită de capetele sale. Descrieţi un algoritm de timp $O(n \lg n)$ pentru determinarea numărului de perechi de coarde care se intersectează în interiorul cercului. (De exemplu, dacă cele n coarde sunt toate diametre, acestea se vor intersecta în centrul cercului şi prin urmare răspunsul corect este $\binom{n}{2}$.) Presupuneţi că nu există două coarde cu acelaşi capăt.

15.2. Cum se îmbogăţeşte o structură de date

La proiectarea algoritmilor apare frecvent problema îmbogăţirii structurilor de date uzuale pentru ca acestea să ofere funcţionalitate suplimentară. Acest proces de îmbogăţire va fi folosit în secţiunea următoare pentru a proiecta o structură de date care are operaţii cu intervale. În această secţiune vom examina paşii acestui proces de îmbogăţire. De asemenea, vom demonstra o teoremă care ne va permite să îmbogăţim mai simplu arborii roşu-negru în multe situaţii.

Procesul de îmbogăţire a unei structuri de date se poate împărţi în patru paşi:

1. alegerea unei structuri de date suport,
2. determinarea informaţiei suplimentare care trebuie inclusă şi întreţinută în structura de date de bază,
3. verificarea posibilităţii întreţinerii informaţiei suplimentare în cazul operaţiilor de bază care modifică structura suport, şi
4. proiectarea noilor operaţii.

Ca şi în cazul altor metode de proiectare prezentate schematic, nu trebuie să respectăm orbeşte paşii de mai sus în ordinea enumerată. Prin natura ei, munca de proiectare recurge frecvent la încercări şi reveniri în caz de eşec, iar realizarea activităţilor prevăzute în paşi se face de regulă în paralel. Nu are nici un rost, de exemplu, să determinăm informaţia suplimentară şi să proiectăm noile operaţii (paşii 2 şi 4) dacă nu vom putea întreţine eficient informaţia suplimentară memorată în structura de date. Totuşi, această metodă în patru paşi va asigura o abordare ordonată a eforturilor de îmbogăţire a unei structuri de date şi o schemă de organizare a documentaţiei referitoare la aceasta.

Paşii de mai sus au fost exemplificaţi în secţiunea 15.1 pentru proiectarea arborilor de statistică de ordine. La pasul 1 am ales ca structură de date suport arborii roşu-negru. O indicaţie asupra utilităţii acestor arbori la obţinerea statisticilor de ordine este dată de eficienţa operaţiilor specifice mulţimilor dinamice pe o ordine totală, precum MINIM, MAXIM, SUCCESOR şi PREDECESOR.

La pasul 2 am adăugat câmpurile *dimensiune*, care conţin, pentru fiecare nod x, dimensiunea subarborelui având rădăcina x. În general, informaţia suplimentară are ca scop să facă operaţiile (proprii structurii îmbogăţite) mai eficiente. De exemplu, am fi putut implementa operaţiile SO-SELECTEAZĂ şi SO-RANG folosind numai cheile memorate în arbore, însă în acest caz timpul lor de execuţie nu ar fi fost $O(\lg n)$. Uneori această informaţie suplimentară conţine referinţe şi nu date, ca în exerciţiul 15.2-1.

Pentru pasul 3, ne-am asigurat că operaţiile de inserare şi ştergere pot întreţine câmpurile *dimensiune* fără a-şi altera eficienţa asimptotică, ele executându-se tot într-un timp $O(\lg n)$. În cazul ideal, un număr mic de modificări ale structurii de date, trebuie să fie suficient pentru a întreţine informaţia suplimentară. De exemplu, dacă am memora în fiecare nod rangul său din arbore, atunci operaţiile SO-SELECTEAZĂ şi SO-RANG s-ar executa mai rapid, dar inserarea unui nou element minim în arbore, ar provoca modificarea acestei informaţii (rangul) în fiecare dintre nodurile arborelui. În schimb, dacă memorăm dimensiunea subarborilor în locul rangului, inserarea unui nou element va necesita modificarea informaţiei suplimentare doar în $O(\lg n)$ noduri.

La pasul 4, am proiectat operaţiile SO-SELECTEAZĂ şi SO-RANG. De fapt, îmbogăţirea structurii de date este impusă tocmai de nevoia de a implementa operaţii noi. Uneori, informaţia suplimentară se va putea folosi nu numai la proiectarea operaţiilor noi, ci şi la reproiectarea celor proprii structurii de date suport, ca în exerciţiul 15.2-1.

Îmbogăţirea arborilor roşu-negru

În cazul în care o structură de date îmbogăţită are la bază arbori roşu-negru, putem să demonstrăm că anumite tipuri de informaţii suplimentare se pot întreţine eficient prin inserare şi ştergere, simplificându-se astfel realizarea pasului 3. Demonstraţia teoremei următoare este similară argumentării făcute în secţiunea 15.1 referitoare la întreţinerea câmpului *dimensiune* pentru arborii de statistică de ordine.

Teorema 15.1 (Îmbogăţirea unui arbore roşu-negru)
Fie T un arbore roşu-negru având n noduri şi fie f un câmp suplimentar al acestuia. Dacă câmpul f din nodul x se poate calcula folosind numai informaţia din nodurile x, $\mathit{stânga}[x]$ şi $dreapta[x]$, inclusiv $f[\mathit{stânga}[x]]$ şi $f[dreapta[x]]$, atunci valorile câmpurilor f din toate nodurile lui T se pot întreţine în timpul operaţiilor de inserare şi ştergere fără a fi afectată performanţa asimptotică de $O(\lg n)$ a acestor operaţii.

Demonstraţie. Ideea principală a demonstraţiei se bazează pe faptul că modificarea câmpului f dintr-un nod x se propagă numai la strămoşii lui x din arbore. Astfel, modificarea lui $f[x]$ s-ar putea să necesite actualizarea lui $f[p[x]]$, şi doar atât; modificarea lui $f[p[x]]$ s-ar putea să necesite actualizarea lui $f[p[p[x]]]$, şi doar atât; şi tot aşa urcăm în arbore câte un nivel la fiecare modificare. Acest proces se va termina când se actualizează $f[\mathit{rădăcină}[T]]$, deoarece nu mai există alt nod care să depindă de noua valoare. Deoarece înălţimea unui arbore roşu-negru este $O(\lg n)$, modificarea unui câmp f dintr-un nod cere un timp de $O(\lg n)$ pentru actualizarea nodurilor dependente de această modificare.

Inserarea unui nod x în T se desfăşoară în două faze (vezi secţiunea 14.3). Pe parcursul primei faze x se inserează ca fiu al unui nod existent $p[x]$. Valoarea lui $f[x]$ se poate calcula într-un timp de $O(1)$ deoarece, prin ipoteză, această valoare depinde numai de informaţia existentă în

celelalte câmpuri ale lui x şi de informaţia din fiii lui x; în acest caz, ambii fii ai lui x sunt NIL. O dată calculată valoarea $f[x]$, modificarea ei se propagă în sus în arbore. Prin urmare, timpul total pentru prima fază a inserării este $O(\lg n)$. În faza a doua singurele modificări structurale ale arborelui provin din rotaţii. Timpul total de actualizare al câmpurilor f este $O(\lg n)$ pentru fiecare rotaţie deoarece într-o rotaţie se modifică doar două noduri. Şi deoarece la o inserare sunt necesare cel mult două rotaţii, timpul total pentru inserare este tot $O(\lg n)$.

La fel ca inserarea, ştergerea are tot două faze (vezi secţiunea 14.4). În prima fază se produc modificări structurale în arbore în două momente: când nodul care se şterge este înlocuit cu succesorul său şi apoi când nodul care se şterge sau succesorul său este eliminat din arbore. Propagarea actualizărilor lui f provocate de aceste modificări necesită cel mult $O(\lg n)$ deoarece modificările sunt locale în arbore. Refacerea arborelui roşu-negru din faza a doua a ştergerii necesită cel mult trei rotaţii, iar fiecare rotaţie necesită un timp cel mult egal cu $O(\lg n)$ pentru a propaga actualizările lui f. În consecinţă, timpul total consumat pentru ştergere este acelaşi cu cel consumat pentru inserare, adică $O(\lg n)$. ■

În multe situaţii, cum ar fi întreţinerea câmpurilor *dimensiune* din arborii de statistică de ordine, costul de actualizare după o rotaţie este $O(1)$ şi nu $O(\lg n)$, estimare folosită în demonstraţia teoremei 15.1. Exerciţiul 15.2-4 conţine un astfel de exemplu.

Exerciţii

15.2-1 Arătaţi cum se pot executa operaţiile (interogările) definite pe mulţimi dinamice MINIM, MAXIM, SUCCESOR şi PREDECESOR, într-un timp $O(1)$, în cazul cel mai defavorabil, pe un arbore îmbogăţit de statistică de ordine. Performanţa asimptotică a celorlalte operaţii pe arborii de statistică de ordine trebuie să rămână aceeaşi.

15.2-2 Studiaţi dacă înălţimile negre ale nodurilor se pot considera câmpuri în nodurile unui arbore, întreţinute corespunzător, fără ca performanţa asimptotică a oricăreia dintre operaţiile arborilor roşu-negru să fie afectată. Arătaţi cum se poate realiza întreţinerea, dacă răspunsul este afirmativ sau justificaţi răspunsul negativ.

15.2-3 Studiaţi dacă adâncimile nodurilor dintr-un arbore roşu-negru se pot considera câmpuri ale nodurilor care se pot întreţine eficient. Arătaţi cum se poate realiza întreţinerea dacă răspunsul este afirmativ sau justificaţi răspunsul negativ.

15.2-4 ⋆ Fie $\otimes$ un operator binar asociativ şi fie a câmpul care trebuie întreţinut în fiecare nod al unui arbore roşu-negru. Presupunem că dorim să adăugăm fiecărui nod x din arbore un câmp suplimentar f astfel încât $f[x] = a[x_1] \otimes a[x_2] \otimes \ldots \otimes a[x_m]$, unde $x_1, x_2, \ldots, x_m$ este lista în inordine a nodurilor din subarborele având rădăcina x. Arătaţi că aceste câmpuri f se pot actualiza corespunzător într-un timp $O(1)$ după o rotaţie. Modificând puţin argumentaţia, arătaţi că şi câmpurile *dimensiune* din arborii de statistică de ordine se pot întreţine într-un timp $O(1)$ pentru fiecare rotaţie.

15.2-5 ⋆ Dorim să îmbogăţim arborii roşu-negru cu o operaţie numită RN-ENUMERĂ(x, a, b) care afişează toate cheile k din arborele roşu-negru având rădăcina x care verifică relaţia $a \leq k \leq b$. Descrieţi modul în care se poate implementa RN-ENUMERĂ într-un timp $\Theta(m + \lg n)$, unde m este numărul de chei care se afişează, iar n este numărul de noduri interne din arbore. (*Indicaţie:* Nu este nevoie să se adauge câmpuri suplimentare la nodurile arborelui.)

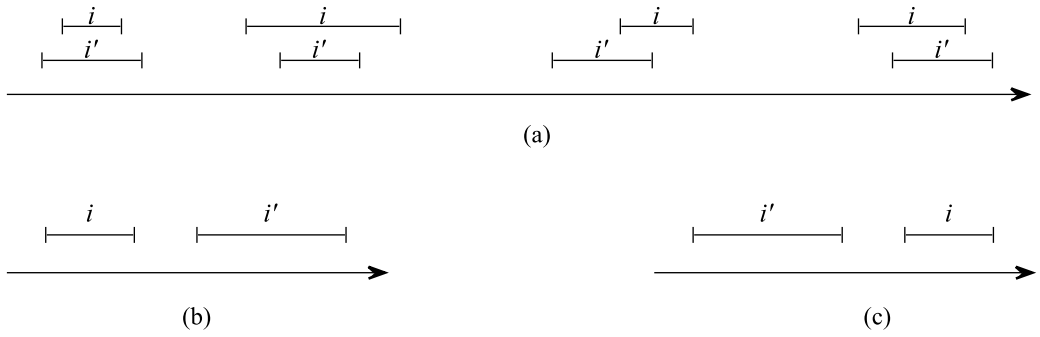


Figura 15.3 Trihotomia intervalelor, pentru două intervale închise i şi i'. **(a)** Când i şi i' se suprapun, sunt patru situaţii posibile; în fiecare dintre ele, $jos[i] \leq sus[i']$ şi $jos[i'] \leq sus[i]$. **(b)** $sus[i] < jos[i']$. **(c)** $sus[i'] < jos[i]$.

15.3. Arbori de intervale

În această secţiune vom îmbogăţi arborii roşu-negru pentru a realiza operaţii pe mulţimi dinamice de intervale. Un ***interval închis*** este o pereche ordonată de numere reale $[t_1, t_2]$ cu $t_1 \leq t_2$. Intervalul $[t_1, t_2]$ reprezintă mulţimea $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$. Intervalele ***deschise*** sau ***semideschise*** nu conţin nici una dintre marginile mulţimii, respectiv nu conţin una dintre acestea. În această secţiune vom presupune că intervalele cu care lucrăm sunt închise; extinderea rezultatelor la intervale deschise sau semideschise este imediată.

Intervalele reprezintă un instrument convenabil de reprezentare a evenimentelor care ocupă fiecare un interval continuu de timp. Am putea dori, de exemplu, să interogăm o bază de date de intervale de timp pentru a descoperi ce evenimente au apărut într-un anumit interval. Structura de date prezentată în această secţiune oferă un mijloc eficient de întreţinere a unei astfel de baze de date de intervale.

Intervalul $[t_1, t_2]$ se poate reprezenta sub forma unui obiect i, care are câmpurile $jos[i] = t_1$ (***capătul inferior***) şi $sus[i] = t_2$ (***capătul superior***). Spunem că intervalele i şi i' se ***suprapun*** dacă $i \cap i' \neq \phi$, adică dacă $jos[i] \leq sus[i']$ şi $jos[i'] \leq sus[i]$. Pentru orice două intervale i şi i' are loc ***trihotomia intervalelor***, adică ele verifică doar una din următoarele trei proprietăţi:

a. i şi i' se suprapun,

b. $sus[i] < jos[i']$,

c. $sus[i'] < jos[i]$.

În figura 15.3 sunt prezentate cele trei posibilităţi.

Numim ***arbore de intervale*** un arbore roşu-negru care întreţine o mulţime dinamică de elemente, în care fiecare element x conţine un interval $int[x]$. Arborii de intervale au următoarele operaţii.

> INTERVAL-INSEREAZĂ(T, x) adaugă la arborele de intervale T elementul x care are în câmpul int un interval.

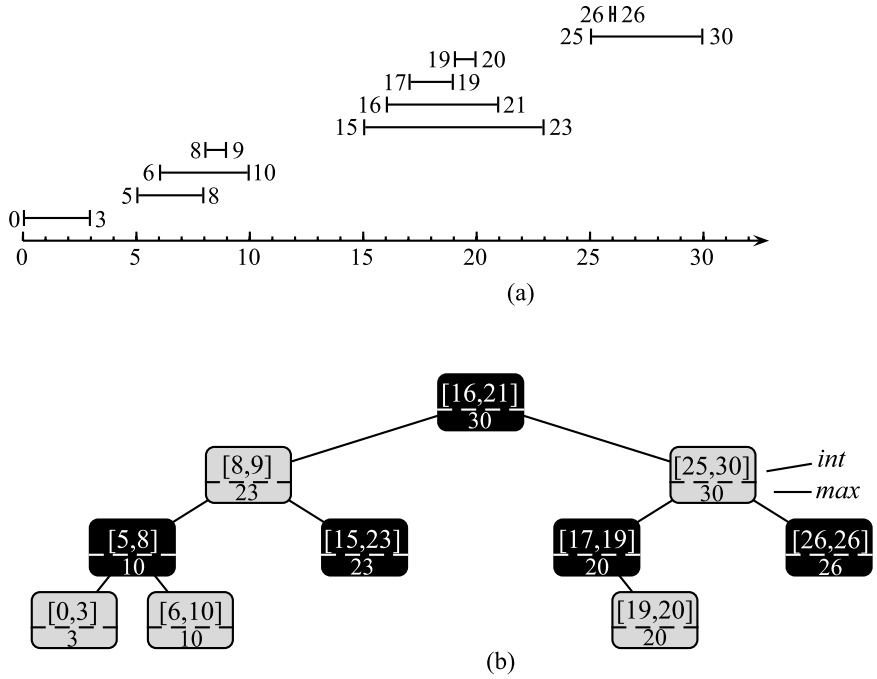


Figura 15.4 Un arbore de intervale. **(a)** O mulţime de 10 intervale, sortate de jos în sus după capătul din stânga (inferior). **(b)** Arborele de intervale care le reprezintă. Traversarea în inordine a acestui arbore va produce lista nodurilor sortată după capătul din stânga.

INTERVAL-ŞTERGE(T, x) şterge elementul x din arborele de intervale T.

INTERVAL-CAUTĂ(T, i) returnează un pointer la un element x din arborele de intervale T pentru care $int[x]$ se suprapune cu intervalul i, sau NIL dacă în mulţime nu există un astfel de element.

Figura 15.4 ilustrează modul de reprezentare al unei mulţimi de intervale cu ajutorul unui arbore de intervale. Vom aplica metoda în patru paşi descrisă în secţiunea 15.2 pentru a proiecta arborele de intervale şi operaţiile proprii acestuia.

Pasul 1: Structura de date suport

Am ales ca structură de date suport un arbore roşu-negru în care fiecare nod x conţine un câmp interval $int[x]$ şi pentru care cheia nodului x este capătul inferior, $jos[int[x]]$, al intervalului. Prin urmare, o traversare în inordine a structurii de date va produce o listă a intervalelor, ordonată după capătul inferior al acestora.

Pasul 2: Informaţia suplimentară

Pe lângă câmpul interval $int[x]$, fiecare nod x va conţine o valoare $max[x]$ care reprezintă valoarea maximă a capetelor intervalelor memorate în subarborele având rădăcina x. Deoarece

capătul superior al fiecărui interval este cel puţin la fel de mare ca şi capătul inferior al acestuia, $max[x]$ va fi de fapt valoarea maximă a capetelor superioare ale intervalelor memorate în subarborele având rădăcina x.

Pasul 3: Întreţinerea informaţiei suplimentare

Trebuie să verificăm că inserarea şi ştergerea se pot executa într-un timp $O(\lg n)$ pe un arbore de intervale având n noduri. Valoarea câmpului $max[x]$ se poate determina dacă se cunosc intervalul $int[x]$ şi valorile max ale fiilor nodului x:

$max[x] = \max(sus[int[x]], max[\textit{stânga}[x]], max[dreapta[x]])$.

Folosind teorema 15.1, rezultă că inserarea şi ştergerea se execută într-un timp $O(\lg n)$. De fapt, actualizarea câmpurilor max după o rotaţie se poate realiza într-un timp $O(1)$, aşa cum se arată în exerciţiile 15.2-4 şi 15.3-1.

Pasul 4: Proiectarea noilor operaţii

Singura operaţie care trebuie proiectată este INTERVAL-CAUTĂ(T, i), care găseşte un interval din T suprapus pe intervalul i. Dacă T nu conţine nici un astfel de interval, se returnează NIL.

```
INTERVAL-CAUTĂ(T, i)
1: x ← rădăcină[T]
2: cât timp x ≠ NIL şi i nu se suprapune cu int[x] execută
3:    dacă stânga[x] ≠ NIL şi max[stânga[x]] ≥ jos[i] atunci
4:       x ← stânga[x]
5:    altfel
6:       x ← dreapta[x]
7: returnează x
```

Căutarea intervalului care se suprapune cu i începe iniţializând x cu rădăcina arborelui şi continuă coborând în arbore. Ea se termină fie când s-a găsit un interval care se suprapune cu i, fie când x devine NIL. Deoarece fiecare iteraţie a ciclului necesită un timp $O(1)$ şi deoarece înălţimea unui arbore roşu-negru având n noduri este $O(\lg n)$, procedura INTERVAL-CAUTĂ va consuma un timp $O(\lg n)$.

Înainte de a studia dacă algoritmul INTERVAL-CAUTĂ este corect, să examinăm cum funcţionează el pe arborele de intervale din figura 15.4. Presupunem că dorim să determinăm un interval care se suprapune cu intervalul $i = [22, 25]$. Se începe iniţializând x cu rădăcina arborelui, care conţine intervalul $[16, 21]$ care nu se suprapune cu i. Deoarece $max[\textit{stânga}[x]] = 23$ este mai mare decât $jos[i] = 22$, ciclul continuă cu x iniţializat cu fiul stâng al rădăcinii – noul nod x care conţine intervalul $[8, 9]$ şi care nici el nu se suprapune cu i. De data aceasta, $max[\textit{stânga}[x]] = 10$ este mai mic decât $jos[i] = 22$, deci ciclul continuă iniţializând x cu fiul său drept. Intervalul $[15, 23]$ memorat în acest nod se suprapune cu i, deci procedura va returna acest nod.

Pentru a da un exemplu de căutare fără succes, să determinăm intervalul care se suprapune cu intervalul $i = [11, 14]$ în arborele de intervale din figura 15.4. Începem iniţializând x cu rădăcina arborelui. Deoarece intervalul memorat în rădăcină nu se suprapune cu i şi $max[\textit{stânga}[x]] = 23$ este mai mare decât $jos[i] = 11$, vom continua cu fiul stâng al lui x, nodul care conţine intervalul $[8, 9]$. (Să observăm că în subarborele drept nu există nici un interval care să se suprapună

cu i – vom vedea mai târziu de ce.) Intervalul din x, [8, 9] nu se suprapune cu i, şi deoarece $max[\mathit{stânga}[x]] = 10$ este mai mic decât $jos[i] = 11$ vom continua iniţializând x cu fiul său drept. (Să observăm acum că în subarborele stâng nu există nici un interval care să se suprapună cu i.) Intervalul din nodul x curent, adică [15, 23] nu se suprapune cu i, fiul său stâng este NIL, aşa că vom continua cu fiul său drept, care este şi el NIL, deci ciclul se încheie şi procedura returnează NIL.

Pentru a studia corectitudinea algoritmului INTERVAL-CAUTĂ, trebuie să înţelegem de ce este suficient să se examineze nodurile de pe un singur drum care porneşte de la rădăcină. Ideea de bază este aceea că pentru orice nod x, dacă $int[x]$ nu se suprapune cu i, atunci căutarea continuă într-o direcţie sigură: intervalul care se suprapune cu i va fi găsit cu siguranţă dacă în arbore există un asemenea interval. Teorema următoare exprimă mai riguros această proprietate.

Teorema 15.2 Considerăm execuţia unei iteraţii a ciclului **cât timp** din algoritmul INTERVAL-CAUTĂ(T, i).

1. Dacă se execută linia 4 (deci căutarea continuă la stânga), atunci fie că subarborele stâng al lui x conţine intervalul care se suprapune cu i, fie că în subarborele drept nu există nici un interval care se suprapune cu i.

2. Dacă se execută linia 6 (deci căutarea continuă la dreapta), atunci în subarborele stâng al lui x nu există nici un interval care se suprapune cu i.

Demonstraţie. Ambele cazuri se demonstrează folosind trihotomia intervalelor. Vom demonstra la început cazul 2, acesta fiind mai simplu. Să remarcăm că dacă se execută linia 6, atunci datorită condiţiei de ramificare din linia 3, avem fie $\mathit{stânga}[x] = \text{NIL}$ fie $max[\mathit{stânga}[x]] < jos[i]$. Dacă $\mathit{stânga}[x] = \text{NIL}$, atunci este evident că subarborele având rădăcina $\mathit{stânga}[x]$ nu conţine nici un interval care să se suprapună cu i deoarece el nu conţine nici un nod, deci nici un interval. Presupunem acum că $\mathit{stânga}[x] \neq \text{NIL}$ şi că $max[\mathit{stânga}[x]] < jos[i]$. Fie i' un interval din subarborele stâng al lui x (vezi figura 15.5(a)). Deoarece $max[\mathit{stânga}[x]]$ este cel mai mare capăt de interval din subarborele stâng al lui x, avem:

$$sus[i'] \quad \leq \quad max[stânga[x]] < jos[i] \ ,$$

şi din trihotomia intervalelor rezultă că i şi i' nu se suprapun, deci am demonstrat cazul 2.

Pentru a demonstra cazul 1, putem presupune că în subarborele stâng al lui x nu există intervale care să se suprapună cu i (dacă ar exista astfel de intervale, am terminat demonstraţia) şi atunci trebuie doar să demonstrăm că nici în subarborele drept al lui x nu există intervale care să se suprapună cu i. Să observăm că dacă se execută linia 4, atunci datorită condiţiei de ramificare din linia 3, avem $max[\mathit{stânga}[x]] \geq jos[i]$. Mai mult, din definiţia câmpului max rezultă că în subarborele stâng al lui x trebuie să existe un interval i' astfel încât:

$$sus[i'] \quad = \quad max[stânga[x]] \geq jos[i].$$

(Figura 15.5(b) reflectă această situaţie). Deoarece i şi i' nu se suprapun şi deoarece relaţia $sus[i'] < jos[i]$ nu este adevărată, din trihotomia intervalelor rezultă că $sus[i] < jos[i']$. Deoarece arborii de intervale au drept chei capetele din stânga ale intervalelor, din proprietatea arborelui de căutare rezultă că pentru orice interval i'' din subarborele stâng al lui x are loc inegalitatea

$$sus[i] \quad < \quad jos[i'] \leq jos[i''].$$

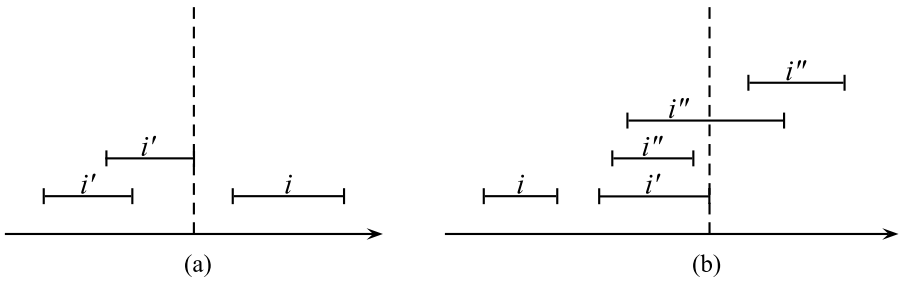


Figura 15.5 Intervalele pentru demonstraţia teoremei 15.2. Valoarea lui $max[stânga[x]]$ este marcată în fiecare caz cu o linie întreruptă. **(a)** Cazul 2: căutarea continuă la dreapta. Nu există nici un interval i' care să se suprapună cu i. **(b)** Cazul 1: căutarea continuă la stânga. Fie că subarborele stâng al lui x conţine un interval care se suprapune cu i (această situaţie nu este ilustrată în figură), fie că există un interval i' în subarborele stâng al lui x astfel ca $sus[i'] = max[stânga[x]]$. Deoarece i nu se suprapune cu i', orice alt interval i'' din subarborele stâng al lui x nu se va suprapune cu i deoarece $jos[i'] \leq jos[i'']$.

Conform trihotomiei intervalelor, rezultă că i şi i'' nu se suprapun. ∎

Teorema 15.2 garantează că dacă algoritmul INTERVAL-CAUTĂ continuă cu unul dintre fiii lui x şi nu se găseşte nici un interval de suprapunere, atunci căutarea care ar continua cu celălalt fiu al lui x ar fi de asemenea fără succes.

Exerciţii

15.3-1 Scrieţi pseudocodul procedurii ROTEŞTE-STÂNGA care operează pe nodurile unui arbore de intervale şi care actualizează câmpurile max într-un timp $O(1)$.

15.3-2 Rescrieţi codul pentru INTERVAL-CAUTĂ astfel ca el să funcţioneze adecvat în cazul când toate intervalele sunt deschise.

15.3-3 Descrieţi un algoritm eficient care, fiind dat un interval i, returnează fie un interval care se suprapune cu i şi care are capătul din stânga minim, fie NIL dacă nu există un asemenea interval.

15.3-4 Fiind daţi un arbore de intervale T şi un interval i, descrieţi cum se poate determina lista tuturor intervalelor din T care se suprapun cu i într-un timp $O(\min(n, k \lg n))$, unde k este numărul de intervale din listă. (*Opţional:* Găsiţi o soluţie care nu modifică arborele.)

15.3-5 Sugeraţi modificări ale procedurilor pentru arborele de intervale pentru a putea implementa operaţia INTERVAL-CAUTĂ-EXACT(T, i), care returnează fie un pointer la un nod x din arborele de intervale pentru care $jos[int[x]] = jos[i]$ şi $sus[int[x]] = sus[i]$, fie NIL dacă T nu conţine un astfel de nod. Toate operaţiile, inclusiv INTERVAL-CAUTĂ-EXACT trebuie să se execute într-un timp de $O(\lg n)$ pe un arbore având n noduri.

15.3-6 Arătaţi cum se poate întreţine o mulţime dinamică Q de numere, care pentru operaţia MIN-DIF returnează mărimea diferenţei dintre cele mai apropiate două numere din Q. De exemplu, dacă $Q = \{1, 5, 9, 15, 18, 22\}$, atunci MIN-DIF$(Q)$ va returna 18 - 15 = 3, deoarece 15

şi 18 sunt cele mai apropiate numere din Q. Realizaţi operaţiile INSEREAZĂ, ŞTERGE, CAUTĂ şi MIN-DIF cât mai eficient şi analizaţi timpii lor de execuţie.

15.3-7 $\star$ Bazele de date de circuite VLSI reprezintă un circuit integrat ca o listă de dreptunghiuri. Presupunem că fiecare dreptunghi este orientat rectiliniu (adică are laturile paralele cu axele x şi y ale sistemului de coordonate), deci reprezentarea unui dreptunghi constă din valorile minime şi maxime ale coordonatelor sale x şi y. Daţi un algoritm ce consumă un timp de $O(n \lg n)$ pentru a decide dacă o mulţime de dreptunghiuri astfel reprezentate conţine sau nu două dreptunghiuri care se suprapun. Algoritmul nu trebuie să determine toate perechile de coordonate care se intersectează, ci doar să raporteze că există suprapunere şi în cazul când un dreptunghi acoperă în întregime alt dreptunghi, chiar dacă laturile lor nu se intersectează. (*Indicaţie:* Deplasaţi o linie de "baleiaj" peste mulţimea de dreptunghiuri.)

Probleme

15-1 Punctul de suprapunere maximă

Presupunem că dorim să memorăm ***punctul de suprapunere maximă*** asociat unei mulţimi de intervale – punctul conţinut în cele mai multe intervale din mulţime. Arătaţi cum se poate întreţine eficient acest punct de suprapunere maximă când se inserează sau se şterg intervale în/din mulţime.

15-2 Permutarea Josephus

Permutarea Josephus se defineşte după cum urmează. Presupunem că n persoane sunt aranjate în cerc şi că există un întreg pozitiv $m \leq n$. Începând cu o persoană stabilită, se parcurge cercul şi fiecare a m-a persoană se scoate din configuraţie. După ce persoana a fost eliminată, numărarea continuă cu configuraţia rămasă. Procesul de eliminare continuă până când toate cele n persoane au fost scoase din configuraţie. Ordinea în care se elimină persoanele din configuraţie va defini ***permutarea Josephus (n,m)*** a întregilor 1, 2, $\ldots, n$. De exemplu, permutarea Josephus (7, 3) este $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

a. Presupunem că m este constant. Descrieţi un algoritm ce consumă un timp $O(n)$ care, fiind dat un întreg n, produce permutarea Josephus (n, m).

b. Presupunem că m nu este constant. Descrieţi un algoritm ce consumă un timp $O(n \lg n)$ care, fiind daţi întregii n şi m, produce permutarea Josephus (n, m).

Note bibliografice

Preparata şi Shamos [160] descriu câteva dintre tipurile de arbori de intervale care apar în literatură. Dintre cei mai importanţi din punct de vedere teoretic amintim aici cei descoperiţi independent de H. Edelsbrunner (1980) şi E.M. McCreight (1981), care, într-o bază de date cu n intervale, permit enumerarea tuturor celor k intervale care suprapun un interval de interogare dat într-un timp $O(k + \lg n)$.

IV Tehnici avansate de proiectare şi analiză

Introducere

Această parte prezintă trei tehnici importante pentru proiectarea şi analiza algoritmilor eficienţi: programarea dinamică (capitolul 16), algoritmii greedy (capitolul 17) şi analiza amortizată (capitolul 18). Părţile anterioare au tratat alte tehnici cu aplicare foarte largă cum ar fi divide şi stăpâneşte, randomizarea şi soluţii recurente. Noile tehnici sunt oarecum mai sofisticate dar sunt esenţiale pentru abordarea eficientă a multor probleme de calcul. Temele introduse în această parte vor reapărea ulterior în carte.

În mod tipic, programarea dinamică este folosită pentru probleme de optimizare în care trebuie făcute un număr finit de alegeri pentru a obţine o soluţie optimală. Deseori, pe măsură ce facem alegerile apar subprobleme de acelaşi tip. Programarea dinamică este eficientă atunci când o subproblemă dată poate apărea din mai multe mulţimi parţiale de alegeri; tehnica cheie este să stocăm sau să "memoizăm" soluţia fiecărei subprobleme în cazul în care aceasta reapare. Capitolul 16 arată cum această idee simplă poate transforma cu uşurinţă algoritmi exponenţiali în algoritmi polinomiali.

La fel ca algoritmii de programare dinamică, algoritmii greedy se folosesc de obicei în probleme de optimizare în care trebuie făcute un număr de alegeri pentru a ajunge la o soluţie optimă. Idea unui algoritm greedy este de a face fiecare alegere într-o manieră optimă local. Un exemplu simplu este plata restului cu număr minim de monede: pentru a minimiza numărul de monede necesare pentru a obţine restul de la o sumă dată de bani este suficient să selectăm în mod repetat moneda cu cea mai mare valoare, dar nu mai mare decât restul rămas. [1] Există multe astfel de probleme pentru care o abordare greedy duce la găsirea unei soluţii optimale mult mai repede decât o abordare prin programare dinamică. Totuşi, nu este întotdeauna uşor de stabilit dacă o abordare greedy va fi eficientă. Capitolul 17 recapitulează teoria matroizilor care poate fi de multe ori folositoare în stabilirea unei astfel de decizii.

Analiza amortizată este un instrument pentru analiza algoritmilor care execută o secvenţă de operaţii similare. În loc să mărginim costul secvenţei de operaţii, mărginind separat costul efectiv al fiecărei operaţii, o analiză amortizată poate fi folosită pentru a furniza o margine a costului efectiv al întregii secvenţe. Un motiv pentru care această idee poate fi eficientă este că, într-o secvenţă de operaţii, s-ar putea să fie imposibil ca toate operaţiile individuale să poată fi efectuate în limitele cunoscute pentru cazul cel mai defavorabil. În timp ce unele operaţii sunt costisitoare, multe altele ar putea să nu fie la fel de costisitoare, din punct de vedere al timpului de execuţie. Analiza amortizată nu este doar un instrument de analiză; ea este şi un mod de gândire referitor la proiectarea algoritmilor deoarece proiectarea unui algoritm şi analiza timpului său de execuţie sunt adesea strâns legate între ele. Capitolul 18 introduce trei modalităţi echivalente de efectuare a unei analize amortizate asupra unui algoritm.

[1] Pentru a fi siguri că restul poate fi întotdeauna plătit, presupunem că există monedă având valoarea egală cu unitatea.

16 Programarea dinamică

Programarea dinamică, asemenea metodelor *divide şi stăpâneşte*, rezolvă problemele combinând soluţiile unor subprobleme. (În acest context, termenul de "programare" se referă la o metodă tabelară şi nu la scrierea codului pentru un calculator.) După cum am arătat în capitolul 1, algoritmii *divide şi stăpâneşte* partiţionează problema în subprobleme independente, rezolvă recursiv subproblemele şi apoi combină soluţiile lor pentru a rezolva problema iniţială. Spre deosebire de această abordare, programarea dinamică este aplicabilă atunci când subproblemele nu sunt independente, adică subproblemele au în comun sub-subprobleme. În acest context, un algoritm de tipul *divide şi stăpâneşte* ar presupune mai multe calcule decât ar fi necesar dacă s-ar rezolva în mod repetat sub-subproblemele comune. Un algoritm bazat pe programare dinamică rezolvă fiecare sub-subproblemă o singură dată şi, apoi, memorează soluţia acesteia într-un tablou, prin aceasta evitând recalcularea soluţiei ori de câte ori respectiva sub-subproblemă apare din nou.

În general, metoda programării dinamice se aplică ***problemelor de optimizare***. Asemenea probleme pot avea mai multe soluţii posibile. Fiecare soluţie are o valoare, iar ceea ce se doreşte este determinarea soluţiei a cărei valoare este optimă (minimă sau maximă). O asemenea soluţie se numeşte *o soluţie* optimă a problemei, prin contrast cu *soluţia* optimă, deoarece pot fi mai multe soluţii care realizează valoarea optimă.

Dezvoltarea unui algoritm bazat pe programarea dinamică poate fi împărţită într-o secvenţă de patru paşi.

1. Caracterizarea structurii unei soluţii optime.

2. Definirea recursivă a valorii unei soluţii optime.

3. Calculul valorii unei soluţii optime într-o manieră de tip "bottom-up".

4. Construirea unei soluţii optime din informaţia calculată.

Paşii 1–3 sunt baza unei abordări de tip programare dinamică. Pasul 4 poate fi omis dacă se doreşte doar calculul unei singure soluţii optime. În vederea realizării pasului 4, deseori se păstrează informaţie suplimentară de la execuţia pasului 3, pentru a uşura construcţia unei soluţii optimale.

Secţiunile care urmează vor folosi metoda programării dinamice pentru a rezolva unele probleme de optimizare. Secţiunea 16.1 abordează problema înmulţirii unui şir de matrice printr-un număr minim de înmulţiri scalare. Pe baza acestui exemplu-problemă abordat prin metoda programării dinamice, secţiunea 16.2 tratează două caracteristici esenţiale pe care trebuie să le prezinte problemele pentru ca programarea dinamică să fie o tehnică de soluţionare viabilă. Secţiunea 16.3 arată cum poate fi determinat cel mai lung subşir comun a două şiruri. În final, secţiunea 16.4 foloseşte programarea dinamică pentru a găsi o triangulare optimă pentru un poligon convex, problemă care este surprinzător de asemănătoare înmulţirii unui şir de matrice.

16.1. Înmulţirea unui şir de matrice

Primul nostru exemplu de programare dinamică este un algoritm care rezolvă problema înmulţirii unui şir de matrice. Se dă un şir (o secvenţă) $\langle A_1, A_2, \ldots, A_n \rangle$ de n matrice care trebuie înmulţite, şi se doreşe calcularea produsului

$$A_1 A_2 \cdots A_n \tag{16.1}$$

Expresia (16.1) se poate evalua folosind algoritmul standard de înmulţire a unei perechi de matrice ca subrutină, o dată ce expresia a fost parantezată (parantezarea este necesară pentru a rezolva toate ambiguităţile privind înmulţirile de matrice). Un produs de matrice este ***complet parantezat*** fie dacă este format dintr-o unică matrice, fie dacă este produsul a două produse de matrice care sunt la rândul lor complet parantezate. Cum înmulţirea matricelor este asociativă, toate parantezările conduc la acelaşi produs. De exemplu, dacă şirul de matrice este $\langle A_1, A_2, A_3, A_4 \rangle$, produsul $A_1 A_2 A_3 A_4$ poate fi complet parantezat în cinci moduri distincte, astfel:

$$(A_1(A_2(A_3A_4))), (A_1((A_2A_3)A_4)), ((A_1A_2)(A_3A_4)), ((A_1(A_2A_3))A_4), (((A_1A_2)A_3)A_4).$$

Modul în care parantezăm un şir de matrice poate avea un impact dramatic asupra costului evaluării produsului. Să considerăm mai întâi costul înmulţirii a două matrice. Algoritmul standard este dat de următoarea procedură, descrisă în pseudocod. Prin *linii* şi *coloane* sunt referite numărul de linii, respectiv de coloane ale matricei.

ÎNMULŢIRE-MATRICE(A, B)

```
1: dacă coloane[A] ≠ linii[B] atunci
2:   mesaj de eroare: "dimensiuni incompatibile"
3: altfel
4:   pentru i ← 1, linii[A] execută
5:     pentru j ← 1, coloane[B] execută
6:       C[i, j] ← 0
7:       pentru k ← 1, coloane[A] execută
8:         C[i, j] ← C[i, j] + A[i, k] · B[k, j]
9:   returnează C
```

Două matrice A şi B se pot înmulţi numai dacă numărul de coloane din A este egal cu numărul de linii din B. Astfel, dacă A este o matrice având dimensiunea $p \times q$ şi B este o matrice având dimensiunea $q \times r$, matricea produsă C este o matrice având dimensiunea $p \times r$. Timpul necesar calculului matricei C este determinat de numărul de înmulţiri scalare (a se vedea linia 8 din algoritm) care este pqr. În cele ce urmează, vom exprima timpii de execuţie în funcţie de numărul de înmulţiri scalare.

Pentru a ilustra modul în care apar costuri diferite la parantezări diferite ale produsului de matrice, să considerăm problema şirului $\langle A_1, A_2, A_3 \rangle$ de trei matrice. Să presupunem că dimensiunile matricelor sunt 10×100, 100×5 şi, respectiv, 5×50. Dacă efectuăm înmulţirle conform parantezării $((A_1A_2)A_3)$, atunci vom avea $10 \cdot 100 \cdot 5 = 5000$ înmulţiri scalare pentru a calcula matricea A_1A_2 de dimensiune 10×5, plus alte $10 \cdot 5 \cdot 50 = 2500$ înmulţiri scalare pentru a înmulţi această matrice cu matricea A_3. Astfel rezultă un total de 7500 înmulţiri

scalare. Dacă, în schimb, vom efectua înmulţirile conform parantezării $(A_1(A_2A_3))$, vom avea $100 \cdot 5 \cdot 50 = 25.000$ înmulţiri scalare pentru a a calcula matricea A_2A_3 de dimensiuni 100×50, plus alte $10 \cdot 100 \cdot 50 = 50.000$ înmulţiri scalare pentru a înmulţi A_1 cu această matrice. Astfel rezultă un total de 75.000 înmulţiri scalare. Deci, calculând produsul conform primei parantezări, calculul este de 10 ori mai rapid.

Problema înmulţirii şirului de matrice poate fi enunţată în următorul mod: dându-se un şir $\langle A_1, A_2, \ldots, A_n \rangle$ de n matrice, unde, pentru $i = 1, 2, \ldots, n$, matricea A_i are dimensiunile $p_{i-1} \times p_i$, să se parantezeze complet produsul $A_1A_2 \cdots A_n$, astfel încât să se minimizeze numărul de înmulţiri scalare.

Evaluarea numărului de parantezări

Înainte de a rezolva problema înmulţirii şirului de matrice prin programare dinamică, trebuie să ne convingem că verificarea exhaustivă a tuturor parantezărilor nu conduce la un algoritm eficient. Fie $P(n)$ numărul de parantezări distincte ale unei secvenţe de n matrice. Deoarece secvenţa de matrice o putem diviza între matricele k şi $(k+1)$, pentru orice $k = 1, 2, \ldots, n-1$, şi, apoi, putem descompune în paranteze, în mod independent, fiecare dintre cele două subsecvenţe, obţinem recurenţa:

$$P(n) = \begin{cases} 1 & \text{dacă } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{dacă } n \geq 2. \end{cases}$$

Problema 13-4 a cerut să se demonstreze că soluţia acestei recurenţe este secvenţa de ***numere Catalan***:

$$P(n) = C(n-1),$$

unde

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2}).$$

Numărul de soluţii este, deci, exponenţial în n, şi metoda forţei brute a căutării exhaustive este, prin urmare, o strategie slabă de determinare a modalităţii de parantezare a şirului de matrice.

Structura unei parantezări optime

Primul pas din schema generală a metodei programării dinamice constă în caracterizarea structurii unei soluţii optimale. Pentru problema înmulţirii şirului de matrice, aceasta este descrisă în continuare. Pentru simplitate, vom adopta convenţia de notare $A_{i..j}$ pentru matricea care rezultă în urma evaluării produsului $A_iA_{i+1} \cdots A_j$. O parantezare optimă a produsului $A_1A_2 \cdots A_n$ împarte produsul între A_k şi A_{k+1} pentru un anumit întreg k din intervalul $1 \leq k < n$. Aceasta înseamnă că, pentru o valoare a lui k, mai întâi calculăm matricele $A_{1..k}$ şi $A_{k+1..n}$ şi, apoi, le înmulţim pentru a produce rezultatul final $A_{1..n}$. Costul acestei parantezări optime este, deci, costul calculului matricei $A_{1..k}$, plus costul calculului matricei $A_{k+1..n}$, plus costul înmulţirii celor două matrice.

Observaţia cheie este că parantezarea subşirului "prefix" $A_1A_2 \cdots A_k$, în cadrul parantezării optime a produsului $A_1A_2 \cdots A_n$, trebuie să fie o parantezare *optimă* pentru $A_1A_2 \cdots A_k$.

De ce? Dacă ar fi existat o modalitate mai puţin costisitoare de parantezare a lui $A_1A_2\cdots A_k$, înlocuirea respectivei parantezări în parantezarea lui $A_1A_2\cdots A_n$ ar produce o altă parantezare pentru $A_1A_2\cdots A_n$, al cărei cost ar fi mai mic decât costul optimului, ceea ce este o contradicţie. O observaţie asemănătoare este valabilă şi pentru parantezarea subşirului $A_{k+1}A_{k+2}\cdots A_n$ în cadrul parantezării optime a lui $A_1A_2\cdots A_n$: aceasta trebuie să fie o parantezare optimă pentru $A_{k+1}A_{k+2}\cdots A_n$.

Prin urmare, o soluţie optimă a unei instanţe a problemei înmulţirii şirului de matrice conţine soluţii optime pentru instanţe ale subproblemelor. Existenţa substructurilor optime în cadrul unei soluţii optime este una dintre caracteristicile cadrului de aplicare a metodei programării dinamice, aşa cum se va arăta în secţiunea 16.2.

O soluţie recursivă

Al doilea pas în aplicarea metodei programării dinamice este definirea valorii unei soluţii optime în mod recursiv, în funcţie de soluţiile optime ale subproblemelor. În cazul problemei înmulţirii şirului de matrice, o subproblemă constă în determinarea costului minim al unei parantezări a şirului $A_iA_{i+1}\cdots A_j$, pentru $1 \le i \le j \le n$. Fie $m[i,j]$ numărul minim de înmulţiri scalare necesare pentru a calcula matricea $A_{i..j}$; costul modalităţii optime de calcul a lui $A_{1..n}$ va fi, atunci, $m[1,n]$.

Putem defini $m[i,j]$ în mod recursiv, după cum urmează. Dacă $i = j$, şirul constă dintr-o singură matrice $A_{i..i} = A_i$ şi, pentru calculul produsului nu este necesară nici o înmulţire scalară. Atunci, $m[i,i] = 0$ pentru $i = 1,2,\ldots,n$. Pentru a calcula $m[i,j]$ când $i < j$, vom folosi structura unei soluţii optimale găsite la pasul 1. Să presupunem că descompunerea optimă în paranteze împarte produsul $A_iA_{i+1}\cdots A_j$ între A_k şi A_{k+1}, cu $i \le k < j$. Atunci, $m[i,j]$ este egal cu costul minim pentru calculul subprodusului $A_{i..k}$ şi $A_{k+1..j}$ plus costul înmulţirii acestor două matrice rezultat. Deoarece evaluarea produsului $A_{i..k}A_{k+1..j}$ necesită $p_{i-1}p_kp_j$ înmulţiri scalare, vom obţine

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

Această ecuaţie recursivă presupune cunoaşterea valorii lui k, lucru care nu este posibil. Există doar $j-i$ valori posibile pentru k, şi anume $k = i, i+1, \ldots, j-1$. Deoarece parantezarea optimă trebuie să folosească una dintre aceste valori pentru k, va trebui să le verificăm pe toate pentru a o putea găsi pe cea mai bună. Atunci, definiţia recursivă a costului minim a parantezării produsului $A_iA_{i+1}\cdots A_j$ devine

$$m[i,j] = \begin{cases} 0 & \text{dacă } i = j, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{dacă } i < j. \end{cases} \tag{16.2}$$

Valorile $m[i,j]$ exprimă costul soluţiilor optime ale subproblemelor. Pentru a putea urmări modul de construcţie a soluţiei optime, să definim $s[i,j]$ care va conţine valoarea k pentru care împărţirea produsului $A_iA_{i+1}\cdots A_j$ produce o parantezare optimă. Aceasta înseamnă că $s[i,j]$ este egal cu valoarea k pentru care $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

Calculul costului optimal

În acest moment, este uşor să scriem un algoritm recursiv bazat pe recurenţa (16.2) care calculează costul minim $m[1,n]$ al produsului $A_1A_2\cdots A_n$. După cum vom vedea în secţiunea

16.2, acest algoritm necesită un timp exponenţial – nu mai bun decât metoda forţei brute folosită pentru verificarea fiecărui mod de descompunere în paranteze a produsului de matrice.

Observaţia importantă care se impune în acest moment este că avem relativ puţine subprobleme: o problemă pentru fiecare alegere a lui i şi j ce satisfac $1 \leq i \leq j \leq n$, adică un total de $\binom{n}{2} + n = \Theta(n^2)$. Un algoritm recursiv poate întâlni fiecare subproblemă de mai multe ori pe ramuri diferite ale arborelui său de recurenţă. Acestă proprietate de suprapunere a subproblemelor este a doua caracteristică a programării dinamice.

În loc să calculăm recursiv soluţia recurenţei (16.2), vom aplica cel de-al treilea pas al schemei generale a metodei programării dinamice şi vom calcula costul optimal cu o abordare "bottom-up". Algoritmul următor presupune că matricea A_i are dimensiunile $p_{i-1} \times p_i$ pentru $i = 1, 2, \ldots, n$. Intrarea este secvenţa $\langle p_0, p_1, \ldots, p_n \rangle$, unde $lungime[p] = n + 1$. Procedura foloseşte un tablou auxiliar $m[1..n, 1..n]$ pentru costurile $m[i, j]$ şi un tablou auxiliar $s[1..n, 1..n]$ care înregistrează acea valoare a lui k pentru care s-a obţinut costul optim în calculul lui $m[i, j]$.

ORDINE-ŞIR-MATRICE(p)
1: $n \leftarrow$ *lungime*$[p] - 1$
2: **pentru** $i \leftarrow 1, n$ **execută**
3: $\quad m[i, i] \leftarrow 0$
4: **pentru** $l \leftarrow 2, n$ **execută**
5: $\quad$ **pentru** $i \leftarrow 1, n - l + 1$ **execută**
6: $\quad\quad j \leftarrow i + l - 1$
7: $\quad\quad m[i, j] \leftarrow \infty$
8: $\quad\quad$ **pentru** $k \leftarrow i, j - 1$ **execută**
9: $\quad\quad\quad q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
10: $\quad\quad\quad$ **dacă** $q < m[i, j]$ **atunci**
11: $\quad\quad\quad\quad m[i, j] \leftarrow q$
12: $\quad\quad\quad\quad s[i, j] \leftarrow k$
13: **returnează** m, s

Algoritmul completează tabloul m într-un mod ce corespunde rezolvării problemei parantezării unor şiruri de matrice de lungime din ce în ce mai mare. Ecuaţia (16.2) arată că $m[i, j]$, costul de calcul al produsului şirului de $j - i + 1$ matrice, depinde doar de costurile calculării produselor şirurilor de mai puţin de $j - i + 1$ matrice. Aceasta înseamnă că, pentru $k = i, i + 1, \ldots, j - 1$, matricea $A_{i..k}$ este un produs de $k - i + 1 < j - i + 1$ matrice, iar matricea $A_{k+1..j}$ este un produs de $j - k < j - i + 1$ matrice.

Algoritmul iniţializează, mai întâi, $m[i, i] \leftarrow 0$ pentru $i = 1, 2, \ldots, n$ (costul minim al şirurilor de lungime 1) în liniile 2–3. Se foloseşte, apoi, recurenţa (16.2) pentru a calcula $m[i, i + 1]$ pentru $i = 1, 2, \ldots, n - 1$ (costul minim al şirurilor de lungime 2) la prima execuţie a ciclului din liniile 4–12. La a doua trecere prin ciclu, se calculează $m[i, i + 2]$ pentru $i = 1, 2, \ldots, n - 2$ (costul minim al şirurilor de lungime 3) etc. La fiecare pas, costul $m[i, j]$ calculat pe liniile 9–12 depinde doar de intrările $m[i, k]$ şi $m[k + 1, j]$ ale tabloului, deja calculate.

Figura 16.1 ilustrează această procedură pe un şir de $n = 6$ matrice. Deoarece am definit $m[i, j]$ doar pentru $i \leq j$, din tabloul m folosim doar porţiunea situată strict deasupra diagonalei principale. Figura arată tablourile rotite astfel încât diagonala principală devine orizontală, iar şirul de matrice este înscris de-a lungul liniei de jos. Folosind acest model, costul minim $m[i, j]$ de înmulţire a subşirului $A_i A_{i+1} \cdots A_j$ de matrice poate fi găsit la intersecţia liniilor care pornesc

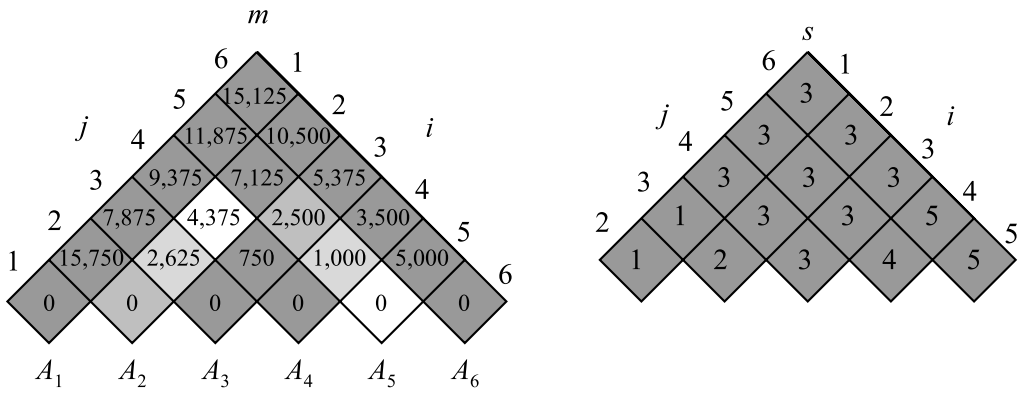


Figura 16.1 Tabelele m şi s calculate de ORDINE-ŞIR-MATRICE pentru $n = 6$ şi matricele cu dimensiunile date mai jos:

matrice	dimensiuni
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Tabelele sunt rotite astfel încât diagonala principală devine orizontală. În tabloul m sunt utilizate doar elementele de deasupra diagonalei principale, iar în tabloul s sunt utilizate doar elementele de sub diagonala principală. Numărul minim de înmulţiri scalare necesare pentru înmulţirea a şase matrice este $m[1,6] = 15.125$. Dintre pătratele haşurate cu gri deschis, perechile care sunt colorate cu aceeaşi nuanţă sunt considerate împreună în linia 9, când calculăm

$$m[2,5] = \min \left\{ \begin{array}{lcll} m[2,2] + m[3,5] + p_1 p_2 p_5 & = & 0 + 2500 + 35 \cdot 15 \cdot 20 & = & 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 & = & 2625 + 1000 + 35 \cdot 5 \cdot 20 & = & 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 & = & 4375 + 0 + 35 \cdot 10 \cdot 20 & = & 11375 \end{array} \right\} = 7125.$$

pe direcţia nord-est din A_i şi pe direcţia nord-vest din A_j. Fiecare rând orizontal din tablou conţine valorile pentru şiruri de matrice de aceeaşi lungime. Procedura ORDINE-ŞIR-MATRICE calculează rândurile de jos în sus şi de la stânga la dreapta în cadrul fiecărui rând. O valoare $m[i,j]$ este calculată pe baza folosirii produselor $p_{i-1} p_k p_j$ pentru $k = i, i+1, \ldots, j-1$ şi a tuturor valorilor aflate pe direcţia sud-est şi sud-vest faţă de $m[i,j]$.

O simplă examinare a structurii de ciclu imbricat din ORDINE-ŞIR-MATRICE conduce la constatarea că timpul de execuţie a algoritmului este $O(n^3)$. În fiecare dintre cele trei cicluri imbricate, indicii acestora (l, i şi k) iau cel mult n valori. Exerciţiul 16.1-3 cere să se demonstreze că timpul de execuţie al acestui algoritm este $\Omega(n^3)$. Algoritmul necesită un spaţiu $\Theta(n^2)$ pentru a stoca tablourile m şi s. De aceea, algoritmul ORDINE-ŞIR-MATRICE este mult mai eficient decât metoda de enumerare a tuturor variantelor de parantezare a produsului de matrice şi de verificare a acestora.

Construirea unei soluţii optime

Deşi algoritmul ORDINE-ŞIR-MATRICE determină numărul optim de înmulţiri scalare necesare pentru calculul produsului şirului de matrice, acesta nu prezintă în mod direct, cum trebuie făcută înmulţirea. Pasul 4 al schemei generale a metodei programării dinamice urmăreşte construirea unei soluţii optime din informaţia disponibilă.

În acest caz particular, vom folosi tabloul $s[1..n, 1..n]$ pentru a determina modul optim de înmulţire a matricelor. Fiecare element $s[i, j]$ conţine valoarea lui k pentru care parantezarea optimă a produsului $A_i A_{i+1} \cdots A_j$ împarte produsul între A_k şi A_{k+1}. Atunci ştim că, în produsul final de calcul al matricei $A_{1..n}$, optimul este $A_{1..s[1,n]} A_{s[1,n]+1..n}$. Înmulţirile anterioare pot fi determinate recursiv, deoarece $s[1, s[1, n]]$ determină ultima înmulţire matriceală din calculul lui $A_{1..s[1,n]}$ şi $s[s[1, n] + 1, n]$ determină ultima înmulţire matriceală din calculul lui $A_{s[1,n]+1..n}$. Următoarea procedură recursivă calculează produsul şirului de matrice $A_{i..j}$, fiind date matricele $A = \langle A_1, A_2, \ldots, A_n \rangle$, tabloul s calculat de ORDINE-ŞIR-MATRICE şi indicii i şi j. Apelul iniţial este ÎNMULŢIRE-ŞIR-MATRICE(A,s,1,n).

ÎNMULŢIRE-ŞIR-MATRICE(A,s,i,j)
1: **dacă** $j > i$ **atunci**
2: $X \leftarrow$ ÎNMULŢIRE-ŞIR-MATRICE$(A, s, i, s[i, j])$
3: $Y \leftarrow$ ÎNMULŢIRE-ŞIR-MATRICE$(A, s, s[i, j] + 1, j)$
4: **returnează** ÎNMULŢIRE-MATRICE(X, Y)
5: **altfel**
6: **returnează** A_i

În exemplul din figura 16.1, apelul ÎNMULŢIRE-ŞIR-MATRICE(A,s,1,6) calculează produsul şirului de matrice conform descompunerii în paranteze

$$((A_1(A_2A_3))((A_4A_5)A_6)). \tag{16.3}$$

Exerciţii

16.1-1 Găsiţi o parantezare optimă a produsului unui şir de matrice al cărui şir de dimensiuni este $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

16.1-2 Proiectaţi un algoritm eficient SCRIE-OPTIM-PĂRINŢI pentru a afişa parantezarea optimă a unui şir de matrice dat de tabloul s calculat de ORDINE-ŞIR-MATRICE. Analizaţi algoritmul.

16.1-3 Fie $R(i, j)$ numărul de accesări ale elementului $m[i, j]$ din tabloul m, de către algoritmul ORDINE-ŞIR-MATRICE, pentru calcularea celorlalte elemente ale tabloului. Arătaţi că, pentru întregul tablou, numărul total de accesări este

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3}.$$

(*Indicaţie:* Identitatea $\sum_{i=1}^{n} i^2 = n(n + 1)(2n + 1)/6$ poate fi utilă.)

16.1-4 Arătaţi că o parantezare completă a unei expresii cu n elemente are exact $n - 1$ perechi de paranteze.

16.2. Elemente de programare dinamică

Cu toate că am prezentat un exemplu de aplicare a metodei programării dinamice, este posibil să existe, în continuare, întrebări asupra situaţiilor în care ea este aplicabilă, sau, altfel spus, când trebuie căutată o soluţie de programare dinamică pentru o anumită problemă? În această secţiune se vor examina două componente de bază pe care trebuie să le prezinte problema de optimizare pentru ca programarea dinamică să fie aplicabilă: substructura optimă şi subproblemele suprapuse. Se va urmări, însă, şi o variantă a metodei, numită memoizare, în care se utilizează proprietatea de suprapunere a subproblemelor.

Substructura optimă

Primul pas al rezolvării unei probleme de optimizare prin programare dinamică constă în caracterizarea structurii unei soluţii optime. Spunem că problema prezintă o ***substructură optimă*** dacă o soluţie optimă a problemei include soluţii optime ale subproblemelor. Ori de câte ori o problemă prezintă o structură optimă, acesta este un bun indiciu pentru posibilitatea aplicării programării dinamice. (De asemenea, aceasta înseamnă că se pot, totuşi, aplica şi strategii greedy; a se vedea capitolul 17).

În secţiunea 16.1 am văzut că problema înmulţirii şirului de matrice prezintă o substructură optimă. Am observat că o parantezare optimă pentru $A_1 A_2 \cdots A_n$, care separă produsul între A_k şi A_{k+1}, include soluţii optime pentru problemele parantezării matricelor $A_1 A_2 \cdots A_k$ şi $A_{k+1} A_{k+2} \cdots A_n$. Tehnica utilizată pentru a arăta că subproblemele au soluţii optime este clasică. Presupunem că există o soluţie mai bună pentru o subproblemă şi arătăm că această presupunere contrazice optimalitatea soluţiei problemei iniţiale.

Substructura optimă a unei probleme sugerează, deseori, un spaţiu potrivit al subproblemelor pentru care se poate aplica programarea dinamică. De obicei, există mai multe clase de subprobleme care pot fi considerate "naturale" pentru o problemă. De exemplu, spaţiul subproblemelor pe care le-am considerat la înmulţirea şirului de matrice conţinea toate subsecvenţele şirului de intrare. La fel de bine, am fi putut alege ca spaţiu al subproblemelor secvenţe arbitrare de matrice din şirul de intrare, dar acest spaţiu al subproblemelor ar fi fost prea mare. Un algoritm de programare dinamică bazat pe acest spaţiu al subproblemelor ar rezolva mult mai multe probleme decât ar fi necesar.

Investigarea substructurii optime a unei probleme prin iterarea instanţelor subproblemelor acesteia este o metodă bună pentru a găsi un spaţiu potrivit de subprobleme pentru programarea dinamică. De exemplu, după examinarea structurii unei soluţii optime a problemei înmulţirii şirului de matrice, putem itera şi examina structura soluţiilor optime ale subproblemelor, a subsubproblemelor şi aşa mai departe. Descoperim, astfel, că toate subproblemele sunt compuse din subsecvenţe ale şirului $\langle A_1, A_2, \ldots, A_n \rangle$. Deci, mulţimea şirurilor de forma $\langle A_i, A_{i+1}, \ldots, A_j \rangle$ cu $1 \leq i \leq j \leq n$ este un spaţiu de subprobleme natural şi rezonabil de utilizat.

Suprapunerea subproblemelor

O a doua componentă pe care trebuie să o aibă o problemă de optimizare pentru ca programarea dinamică să fie aplicabilă este ca spaţiul subproblemelor să fie "restrâns", în sensul că un algoritm recursiv rezolvă mereu aceleaşi subprobleme, în loc să genereze subprobleme noi. De

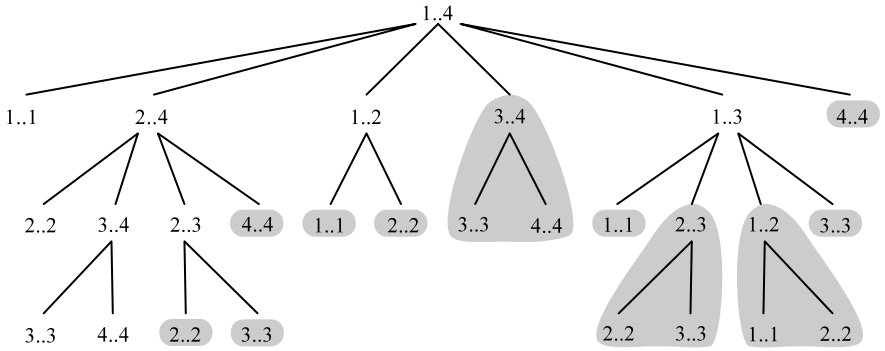


Figura 16.2 Arborele de recurenţă pentru calculul apelului ŞIR-MATRICE-RECURSIV$(p, 1, 4)$. Fiecare nod conţine parametrii i şi j. Calculele efectuate în subarborii haşuraţi sunt înlocuite de apariţia unui tablou de echivalenţă în ŞIR-MATRICE-MEMORAT$(p, 1, 4)$.

obicei, numărul total de subprobleme distincte este dependent polinomial de dimensiunea datelor de intrare. Când un algoritm recursiv abordează mereu o aceeaşi problemă, se spune că problema de optimizare are ***subprobleme suprapuse***. Spre deosebire de aceasta, o problemă care se rezolvă cu un algoritm de tip divide şi stăpâneşte generează, la fiecare nouă etapă, probleme noi. În general, algoritmii de programare dinamică folosesc suprapunerea subproblemelor prin rezolvarea o singură dată a fiecărei subprobleme, urmată de stocarea soluţiei într-un tablou unde poate fi regăsită la nevoie, necesitând pentru regăsire un timp constant.

Pentru a ilustra proprietatea de subprobleme suprapuse, vom reexamina problema înmulţirii şirului de matrice. Revenind la figura 16.1, observăm că algoritmul ORDINE-ŞIR-MATRICE preia, în mod repetat, soluţiile subproblemelor din liniile de jos atunci când rezolvă subproblemele din liniile superioare. De exemplu, valoarea $m[3, 4]$ este referită de patru ori: în timpul calculelor pentru $m[2, 4]$, $m[1, 4]$, $m[3, 5]$ şi $m[3, 6]$. Dacă $m[3, 4]$ ar fi recalculată de fiecare dată în loc să fie preluată din tablou, creşterea timpului de execuţie ar fi dramatică. Pentru a vedea aceasta, să considerăm următoarea procedură recursivă (ineficientă) care determină $m[i, j]$, numărul minim de înmulţiri scalare necesare pentru a calcula produsul şirului de matrice $A_{i..j} = A_i A_{i+1} \dots A_j$. Procedura este bazată direct pe recurenţa (16.2).

```
ŞIR-MATRICE-RECURSIV(p, i, j)
1: dacă i = j atunci
2:    returnează 0
3: m[i, j] ← ∞
4: pentru k ← i, j − 1 execută
5:    q ← ŞIR-MATRICE-RECURSIV(p, i, k) + ŞIR-MATRICE-RECURSIV(p, k + 1, j) + p_{i−1} p_k p_j
6:    dacă q < m[i, j] atunci
7:       m[i, j] ← q
8: returnează m[i, j]
```

Figura 16.2 prezintă arborele de recurenţă produs de apelul ŞIR-MATRICE-RECURSIV$(p,1,4)$. Fiecare nod este etichetat cu valorile parametrilor i şi j. Se observă că anumite perechi de valori apar de mai multe ori.

De fapt, putem arăta că timpul de execuţie $T(n)$ necesar pentru calculul lui $m[1,n]$ prin această procedură recursivă este cel puţin exponenţial în n. Să presupunem că executarea liniilor 1–2 şi a liniilor 6–7 ia cel puţin un timp egal cu unitatea. Inspectarea procedurii conduce la recurenţa

$$\begin{aligned} T(1) &\geq 1, \\ T(n) &\geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1), \text{ pentru } n > 1. \end{aligned}$$

Să observăm că pentru $i = 1, 2, \ldots, n-1$, fiecare termen $T(i)$ apare o dată ca $T(k)$ şi o dată ca $T(n-k)$; adunând cele $n-1$ unităţi cu unitatea din primul termen al sumei, recurenţa se poate rescrie ca

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n. \tag{16.4}$$

Vom arăta că $T(n) = \Omega(2^n)$ folosind inducţia matematică. Mai precis, vom arăta că $T(n) \geq 2^{n-1}$ pentru orice $n \geq 1$. Pentru $n = 1$, este clar că $T(1) \geq 1 = 2^0$. Prin inducţie, pentru $n \geq 2$ avem

$$T(n) \geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n = 2 \sum_{i=0}^{n-2} 2^i + n = 2(2^{n-1} - 1) + n = (2^n - 2) + n \geq 2^{n-1},$$

ceea ce completează demonstraţia. Atunci, totalul calculelor realizate de apelul ŞIR-MATRICE-RECURSIV$(p, 1, n)$ este cel puţin exponenţial în n.

Comparaţi acest algoritm recursiv, "top-down", cu algoritmul de programare dinamică "bottom-up". Acesta din urmă este mult mai eficient deoarece foloseşte proprietatea de suprapunere a subproblemelor. Problema are exact $\Theta(n^2)$ subprobleme distincte, şi algoritmul de programare dinamică o rezolvă pe fiecare o singură dată. Pe de altă parte, algoritmul recursiv trebuie să rezolve în mod repetat fiecare subproblemă, ori de câte ori apare în arborele de recurenţă. Se poate încerca aplicarea programării dinamice ori de câte ori arborele de recurenţă corespunzător soluţiei recursive naturale conţine, în mod repetat, o aceeaşi subproblemă, iar numărul total de subprobleme este mic.

Memoizarea

Vom prezenta în continuare o variantă a programării dinamice, care măreşte, deseori, eficienţa abordării uzuale a programării dinamice, folosind în acelaşi timp o strategie "top-down". Ideea este de a ***memoiza*** algoritmul recursiv natural, dar despre care am arătat că este ineficient. Ca şi în cazul programării dinamice obişnuite, soluţiile subproblemelor se păstrează într-un tablou, dar structura de control pentru completarea tabloului este mult mai asemănătoare algoritmului recursiv.

Un algoritm recursiv memoizat foloseşte un element al tabloului pentru fiecare soluţie a fiecărei subprobleme. Fiecare element al tabloului conţine, la început, o valoare specială care indică faptul că respectivul element trebuie completat. Atunci când subproblema este întâlnită prima oară pe parcursul execuţiei algoritmului recursiv, soluţia sa este calculată şi apoi stocată

în tablou. După aceasta, ori de câte ori este întâlnită subproblema respectivă, valoarea stocată în tablou este căutată şi regăsită.[1]

Procedura următoare este versiunea memoizată a procedurii ŞIR-MATRICE-RECURSIV.

```
ŞIR-MATRICE-MEMOIZAT(p)
1: n ← lungime[p] − 1
2: pentru i ← 1, n execută
3:    pentru j ← i, n execută
4:       m[i, j] ← ∞
5: returnează ŞIR-ECHIVALENT(p, 1, n)
```

```
ŞIR-ECHIVALENT(p, i, j)
1: dacă m[i, j] < ∞ atunci
2:    returnează m[i, j]
3: dacă i = j atunci
4:    m[i, j] ← 0
5: altfel
6:    pentru k ← i, j − 1 execută
7:       q ← ŞIR-ECHIVALENT(p, i, k) + ŞIR-ECHIVALENT(p, k + 1, j) + p_{i−1} p_k p_j
8:    dacă q < m[i, j] atunci
9:       m[i, j] ← q
10: returnează m[i, j]
```

Algoritmul ŞIR-MATRICE-MEMOIZAT, la fel ca şi ORDINE-ŞIR-MATRICE, utilizează tabloul $m[1..n, 1..n]$ al valorilor calculate pentru $m[i, j]$, numărul minim de înmulţiri scalare necesare pentru a calcula matricea $A_{i..j}$. Fiecare element al tabloului conţine, la început, elementul ∞ pentru a arăta că respectivul element trebuie determinat. Atunci când se execută apelul ŞIR-ECHIVALENT(p, i, j), dacă $m[i, j] < \infty$, pe linia 1, procedura se rezumă la a returna costul deja calculat $m[i, j]$ (linia 2). Altfel, costul este calculat la fel ca în algoritmul ŞIR-MATRICE-RECURSIV, memorat în $m[i, j]$ şi returnat. (Valoarea ∞ este convenabilă pentru marcarea elementelor necalculate ale tabloului, deoarece este valoarea folosită la iniţializarea valorii $m[i, j]$ în linia 3 din ŞIR-MATRICE-RECURSIV.) Atunci, ŞIR-ECHIVALENT(p, i, j) returnează, întotdeauna, valoarea $m[i, j]$, dar o calculează efectiv numai în cazul primului apel având parametrii i şi j.

Figura 16.2 ilustrează modul în care algoritmul ŞIR-MATRICE-MEMOIZAT economiseşte timp în raport cu ŞIR-MATRICE-RECURSIV. Subarborii haşuraţi reprezintă valori căutate în tablou şi care nu sunt calculate.

Asemănător algoritmului de programare dinamică ORDINE-ŞIR-MATRICE, procedura ŞIR-MATRICE-MEMOIZAT are timpul de execuţie $O(n^3)$. Fiecare dintre cele $\Theta(n^2)$ elemente ale tabloului este iniţializat o singură dată în linia 4 a procedurii ŞIR-MATRICE-MEMOIZAT şi completată cu valoarea definitivă printr-un singur apel al procedurii ŞIR-ECHIVALENT. Fiecare dintre aceste $\Theta(n^2)$ apeluri durează un timp $O(n)$, fără a lua în considerare timpul necesar calculării celorlalte elemente ale tabloului, deci, în total, este folosit un timp $O(n^3)$. Prin urmare, memoizarea transformă un algoritm $\Omega(2^n)$ într-un algoritm $O(n^3)$.

[1] Această abordare presupune că mulţimea parametrilor tuturor subproblemelor posibile este cunoscută şi că este stabilită relaţia dintre poziţiile din tablou şi subprobleme. O altă abordare presupune memorarea indexată (hashing), folosind parametrii subproblemelor drept chei de căutare.

În concluzie, problema înmulţirii şirului de matrice se poate rezolva în timp $O(n^3)$ fie printr-un algoritm "top-down" cu memoizare, fie printr-un algoritm de programare dinamică "bottom-up". Ambele metode folosesc proprietatea subproblemelor suprapuse. În total există numai $\Theta(n^2)$ subprobleme distincte şi fiecare dintre metodele amintite rezolvă fiecare subproblemă o singură dată. Fără memoizare, algoritmul recursiv natural necesită un timp exponenţial, deoarece subproblemele sunt rezolvate în mod repetat.

În practică, dacă toate subproblemele trebuie rezolvate cel puţin o dată, un algoritm de programare dinamică "bottom-up" îmbunătăţeşte cu un factor constant un algoritm "top-down" cu memoizare, deoarece nu apare efortul suplimentar necesitat de recurenţă, iar pentru gestionarea tabloului, efortul suplimentar este mai mic. Mai mult chiar, există probleme pentru care modul regulat de accesare a tabloului în metoda programării dinamice poate fi folosit pentru a reduce suplimentar timpul sau spaţiul necesar. Reciproc, dacă unele subprobleme din spaţiul subproblemelor nu trebuiesc rezolvate deloc, soluţia cu memoizare are avantajul de a rezolva numai subproblemele necesare.

Exerciţii

16.2-1 Comparaţi recurenţa (16.4) cu recurenţa (8.4), apărută în analiza timpului mediu de execuţie al algoritmului de sortare rapidă. Explicaţi, intuitiv, de ce soluţiile celor două recurenţe sunt atât de diferite.

16.2-2 Care este modul cel mai eficient de determinare a numărului optim de înmulţiri în problema produsului de matrice: enumerarea tuturor parantezărilor posibile ale produsului şi calculul numărului de înmulţiri pentru fiecare descompunere, sau execuţia procedurii ŞIR-MATRICE-RECURSIV? Justificaţi răspunsul.

16.2-3 Desenaţi arborele de recurenţă pentru procedura SORTEAZĂ-PRIN-INTERCLASARE, descrisă în secţiunea 1.3.1, pentru un vector de 16 elemente. Explicaţi de ce memoizarea este ineficientă pentru accelerarea unui algoritm bun de tip divide şi stăpâneşte, aşa cum este SORTEAZĂ-PRIN-INTERCLASARE.

16.3. Cel mai lung subşir comun

Următoarea problemă pe care o vom lua în considerare este problema celui mai lung subşir comun. Un subşir al unui şir dat este şirul iniţial din care lipsesc unele elemente (eventual nici unul). În mod formal, dându-se şirul $X = \langle x_1, x_2, \ldots, x_m \rangle$, un alt şir $Z = \langle z_1, z_2, \ldots, z_k \rangle$ este un ***subşir*** al lui X dacă există un şir strict crescător de indici din X, $\langle i_1, i_2, \ldots, i_k \rangle$, astfel încât, pentru toate valorile $j = 1, 2, \ldots, k$, avem $x_{i_j} = z_j$. De exemplu, $Z = \langle B, C, D, B \rangle$ este un subşir al lui $X = \langle A, B, C, B, D, A, B \rangle$, secvenţa de indici corespunzătoare fiind $\langle 2, 3, 5, 7 \rangle$.

Dându-se două şiruri X şi Y, spunem că şirul Z este un ***subşir comun*** pentru X şi Y dacă Z este un subşir atât pentru X cât şi pentru Y. De exemplu, dacă $X = \langle A, B, C, B, D, A, B \rangle$ şi $Y = \langle B, D, C, A, B, A \rangle$, şirul $\langle B, C, A \rangle$ este un subşir comun pentru X şi Y. Şirul $\langle B, C, A \rangle$ nu este *cel mai lung* subşir comun (CMLSC) pentru X şi Y, deoarece are lungimea 3, iar şirul $\langle B, C, B, A \rangle$, care este de asemenea subşir comun lui X şi Y, are lungimea 4. Şirul $\langle B, C, B, A \rangle$

este un CMLSC pentru X şi Y, la fel şi şirul $\langle B, D, A, B\rangle$, deoarece nu există un subşir comun de lungime cel puţin 5.

În problema ***celui mai lung subşir comun***, se dau două şiruri $X = \langle x_1, x_2, \ldots, x_m\rangle$ şi $Y = \langle y_1, y_2, \ldots, y_n\rangle$ şi se doreşte determinarea unui subşir comun de lungime maximă pentru X şi Y. Această secţiune va arăta că problema CMLSC poate fi rezolvată eficient prin metoda programării dinamice.

Caracterizarea celui mai lung subşir comun

O abordare prin metoda forţei brute pentru rezolvarea problemei CMLSC constă în enumerarea tuturor subşirurilor lui X şi verificarea dacă acestea constituie un subşir şi pentru Y, memorând cel mai lung subşir găsit. Fiecare subşir al lui X corespunde unei submulţimi a indicilor lui X, $\langle 1, 2, \ldots, m\rangle$. Există 2^m subşiruri pentru X şi, deci, această abordare necesită un timp exponenţial, ceea ce o face inabordabilă pentru şiruri de lungimi mari.

Problema CMLSC are proprietatea de substructură optimă, aşa cum vom arăta în următoarea teoremă. Clasa naturală de subprobleme corespunde unor perechi de "prefixe" ale celor două şiruri de intrare. Mai precis, dându-se un şir $X = \langle x_1, x_2, \ldots, x_m\rangle$, definim ***prefixul*** i al lui X, pentru $i = 0, 1, \ldots, m$, ca fiind $X_i = \langle x_1, x_2, \ldots, x_i\rangle$. De exemplu, dacă $X = \langle A, B, C, B, D, A, B\rangle$, atunci $X_4 = \langle A, B, C, B\rangle$ iar X_0 este şirul vid.

Teorema 16.1 (Substructura optimală a unui CMLSC) Fie şirurile $X = \langle x_1, x_2, \ldots, x_m\rangle$ şi $Y = \langle y_1, y_2, \ldots, y_n\rangle$ şi fie $Z = \langle z_1, z_2, \ldots, z_k\rangle$ un CMLSC pentru X şi Y.

1. Dacă $x_m = y_n$, atunci $z_k = x_m = y_n$ şi Z_{k-1} este un CMLSC pentru X_{m-1} şi Y_{n-1}.

2. Dacă $x_m \neq y_n$, atunci, din $z_k \neq x_m$ rezultă că Z este un CMLSC pentru X_{m-1} şi Y.

3. Dacă $x_m \neq y_n$, atunci, din $z_k \neq y_n$ rezultă că Z este un CMLSC pentru X şi Y_{n-1}.

Demonstraţie. (1) Dacă $z_k \neq x_m$, atunci putem adăuga $x_m = y_n$ la Z pentru a obţine un subşir comun a lui X şi Y de lungime $k + 1$, contrazicând presupunerea că Z este *cel mai lung* subşir comun pentru X şi Y. Deci, va trebui să avem $z_k = x_m = y_n$. Prefixul Z_{k-1} este un subşir comun de lungime $k - 1$ pentru X_{m-1} şi Y_{n-1}. Dorim să demonstrăm că este un CMLSC. Să presupunem, prin absurd, că există un subşir W comun pentru X_{m-1} şi Y_{n-1}, a cărui lungime este fie mai mare decât $k - 1$. Atunci, adăugând la W elementul $x_m = y_n$, se va forma un subşir comun a lui X şi Y a cărui lungime este mai mare decât k, ceea ce este o contradicţie.

(2) Dacă $z_k \neq x_m$, atunci Z este un subşir comun pentru X_{m-1} şi Y. Dacă ar exista un subşir comun W al lui X_{m-1} şi Y, cu lungime mai mare decât k, atunci W ar fi şi un subşir comun al lui X_m şi Y, contrazicând presupunerea că Z este un CMLSC pentru X şi Y.

(3) Demonstraţia este simetrică celei de la (2). ∎

Caracterizarea din teorema 16.1 arată că un CMLSC al două şiruri conţine un CMLSC pentru prefixele celor două şiruri. Atunci, problema CMLSC are proprietatea de substructură optimală. O soluţie recursivă are, de asemenea, proprietatea de suprapunere a problemelor, după cum vom arăta în cele ce urmează.

O soluţie recursivă a subproblemelor

Teorema 16.1 implică faptul că există fie una, fie două probleme care trebuie examinate pentru găsirea unui CMLSC pentru $X = \langle x_1, x_2, \ldots, x_m \rangle$ şi $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Dacă $x_m = y_n$, trebuie găsit un CMLSC pentru X_{m-1} şi Y_{n-1}. Adăugarea elementului $x_m = y_n$ la acest CMLSC produce un CMLSC pentru X şi Y. Dacă $x_m \neq y_n$, atunci trebuie rezolvate două subprobleme: găsirea unui CMLSC pentru X_{m-1} şi Y şi găsirea unui CMLSC pentru X şi Y_{n-1}. Cel mai lung CMLSC dintre acestea două va fi CMLSC pentru X şi Y.

Se poate observa că problema CMLSC se descompune în subprobleme suprapuse. Pentru a găsi un CMLSC al şirurilor X şi Y, va trebui, probabil, calculat CMLSC pentru X şi Y_{n-1}, respectiv, pentru X_{m-1} şi Y. Dar fiecare dintre aceste subprobleme conţine sub-subproblema găsirii CMLSC pentru X_{m-1} şi Y_{n-1}. Multe alte subprobleme au în comun sub-subprobleme.

Ca şi problema înmulţirii şirului de matrice, soluţia recursivă pentru problema CMLSC implică stabilirea unei recurenţe pentru costul unei soluţii optime. Să definim $c[i, j]$ ca lungimea unui CMLSC al şirurilor X_i şi Y_j. Dacă $i = 0$ sau $j = 0$, CMLSC are lungimea 0. Substructura optimală a problemei CMLSC produce formula recursivă

$$c[i,j] = \begin{cases} 0 & \text{dacă } i = 0 \text{ sau } j = 0, \\ c[i-1, j-1] + 1 & \text{dacă } i, j > 0 \text{ şi } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{dacă } i, j > 0 \text{ şi } x_i \neq y_j. \end{cases} \tag{16.5}$$

Calculul lungimii unui CMLSC

Pe baza ecuaţiei (16.5) se poate scrie un algoritm recursiv având timp exponenţial pentru calculul lungimii unui CMLSC a două şiruri. Deoarece există numai $\Theta(mn)$ subprobleme distincte, vom folosi metoda programării dinamice pentru a calcula soluţiile în mod "bottom-up".

```
LUNGIME-CMLSC(X, Y)
 1: m ← lungime[X]
 2: n ← lungime[Y]
 3: pentru i ← 1, m execută
 4:    c[i, 0] ← 0
 5: pentru j ← 0, n execută
 6:    c[0, j] ← 0
 7: pentru i ← 1, m execută
 8:    pentru j ← 1, n execută
 9:       dacă x_i = y_j atunci
10:          c[i, j] ← c[i − 1, j − 1] + 1
11:          b[i, j] ←"↖"
12:       altfel
13:          dacă c[i − 1, j] ≥ c[i, j − 1] atunci
14:             c[i, j] ← c[i − 1, j]
15:             b[i, j] ← "↑"
16:          altfel
17:             c[i, j] ← c[i, j − 1]
18:             b[i, j] ← "←"
19: returnează c, b
```

Procedura LUNGIME-CMLSC are ca date de intrare două şiruri $X = \langle x_1, x_2, \ldots, x_m \rangle$ şi $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Procedura memorează valorile $c[i, j]$ într-un tablou $c[0..m, 0..n]$ ale cărui elemente sunt calculate în ordinea crescătoare a liniilor. (Aceasta înseamnă că se completează mai întâi prima linie de la stânga la dreapta, apoi a doua linie şi aşa mai departe.) De asemenea, se construieşte un tablou $b[1..m, 1..n]$ care simplifică determinarea unei soluţii optimale. Intuitiv, $b[i, j]$ indică elementul tabloului care corespunde soluţiei optime a subproblemei alese la calculul lui $c[i, j]$. Procedura returnează tablourile b şi c; $c[m, n]$ conţine lungimea unui CMLSC pentru X şi Y.

Figura 16.3 conţine tabloul produs de LUNGIME-CMLSC pentru şirurile $X = \langle A, B, C, B, D, A, B \rangle$ şi $Y = \langle B, D, C, A, B, A \rangle$. Timpul de execuţie al procedurii este $O(mn)$, deoarece calculul fiecărui element al tabloului necesită un timp $O(1)$.

Construirea unui CMLSC

Tabelul b returnat de LUNGIME-CMLSC poate fi folosit pentru construcţia rapidă a unui CMLSC pentru $X = \langle x_1, x_2, \ldots, x_m \rangle$ şi $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Pur şi simplu, se începe cu $b[m, n]$ şi se parcurge tabloul conform direcţiilor indicate. Ori de câte ori se întâlneşte un element "$\nwarrow$" pe poziţia $b[i, j]$, aceasta înseamnă că $x_i = y_j$ este un element al CMLSC. Prin această metodă, elementele CMLSC sunt regăsite în ordine inversă. Următoarea procedură recursivă tipăreşte un CMLSC al şirurilor X şi Y în ordinea corectă. Apelul iniţial este SCRIE-CMLSC(b,X,$lungime[X]$,$lungime[Y]$).

```
SCRIE-CMLSC(b, X, i, j)
1: dacă i = 0 sau j = 0 atunci
2:    revenire
3:    dacă b[i, j] ="↖" atunci
4:       SCRIE-CMLSC(b, X, i − 1, j − 1)
5:       tipăreşte x_i
6:    altfel dacă b[i, j] ="↑" atunci
7:       SCRIE-CMLSC(b, X, i − 1, j)
8:    altfel
9:       SCRIE-CMLSC(b, X, i, j − 1)
```

Pentru tabloul b din figura 16.3, această procedură tipăreşte "$BCBA$". Procedura necesită un timp de $O(m + n)$, deoarece, cel puţin unul dintre i şi j este decrementat în fiecare etapă a recurenţei.

Îmbunătăţirea codului

O dată ce s-a construit un algoritm, se descoperă că, deseori, timpul de execuţie sau spaţiul folosit pot fi îmbunătăţite. Acest lucru este adevărat, mai ales, pentru programarea dinamică clasică. Anumite modificări pot simplifica implementarea şi pot îmbunătăţi factorii constanţi, dar nu pot aduce o îmbunătăţire asimptotică a performanţei. Alte modificări pot aduce îmbunătăţiri asimptotice substanţiale atât pentru timpul de execuţie cât şi pentru spaţiul ocupat. De exemplu, se poate elimina complet tabloul b. Fiecare element $c[i, j]$ depinde de numai trei alte elemente ale tabloului c: $c[i-1, j-1]$, $c[i-1, j]$ şi $c[i, j-1]$. Dându-se valoarea lui $c[i, j]$, se poate determina, în timp $O(1)$, care dintre aceste trei valori a fost folosită pentru calculul lui $c[i, j]$ fără a mai folosi

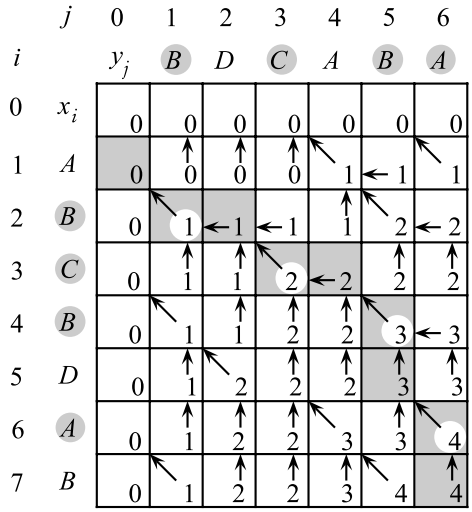


Figura 16.3 Tabelele c şi b calculate de LUNGIME-CMLSC pentru şirurile $X = \langle A, B, C, B, D, A, B\rangle$ şi $Y = \langle B, D, C, A, B, A\rangle$. Pătratul din linia i şi coloana j conţine valoarea $c[i, j]$ precum şi săgeata potrivită pentru valoarea lui $b[i, j]$. Valoarea 4 a lui $c[7, 6]$ – colţul cel mai din dreapta jos în tablou – este lungimea unui CMLSC $\langle B, C, B, A\rangle$ al lui X şi Y. Pentru $i, j > 0$, valoarea $c[i, j]$ depinde doar de valoarea expresiei $x_i = y_j$ şi de valorile elementelor $c[i-1, j]$, $c[i, j-1]$ şi $c[i-1, j-1]$, care sunt calculate înaintea lui $c[i, j]$. Pentru a reconstrui elementele unui CMLSC, se urmăresc săgeţile corespunzătoare lui $b[i, j]$ începând cu colţul cel mai din dreapta; drumul este haşurat. Fiecare "$\nwarrow$" pe acest drum corespunde unui element (care este evidenţiat) pentru care $x_i = y_j$ aparţine unui CMLSC.

tabloul b. Ca o consecinţă, putem reconstrui un CMLSC în timp $O(m + n)$ folosind o procedură similară cu SCRIE-CMLSC. (Exerciţiul 16.3-2 cere algoritmul corespunzător în pseudocod.) Deşi prin această metodă se eliberează un spaţiu $\Theta(mn)$, spaţiul auxiliar necesar calculului unui CMLSC nu descreşte asimptotic, deoarece este oricum necesar un spaţiu $\Theta(mn)$ pentru tabloul c.

Totuşi, spaţiul necesar pentru LUNGIME-CMLSC îl putem reduce deoarece, la un moment dat, sunt necesare doar două linii ale tabloului c: linia care este calculată şi linia precedentă. (De fapt, putem folosi doar puţin mai mult spaţiu decât cel necesar unei linii a lui c, pentru a calcula lungimea unui CMLSC; vezi exerciţiul 16.3-4.) Această îmbunătăţire funcţionează doar dacă este necesară numai lungimea unui CMLSC; dacă este nevoie şi de reconstruirea elementelor unui CMLSC, tabloul mai mic nu conţine destulă informaţie pentru a reface etapele în timp $O(m + n)$.

Exerciţii

16.3-1 Determinaţi un CMLSC pentru $\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$ şi $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0\rangle$.

16.3-2 Arătaţi cum poate fi reconstituit un CMLSC pe baza tabloului c (completat) şi a şirurilor iniţiale $X = \langle x_1, x_2, \ldots, x_m\rangle$ şi $Y = \langle y_1, y_2, \ldots, y_n\rangle$ în timp $O(m + n)$, fără a folosi tabloul b.

16.3-3 Daţi o versiune cu memoizare a procedurii LUNGIME-CMLSC, care se execută în timp $O(mn)$.

16.3-4 Arătaţi cum se poate calcula lungimea unui CMLSC folosind doar $2\min(m, n)$ elemente din tabloul c şi un spaţiu suplimentar de $O(1)$. Indicaţi apoi modul de realizare a acestui calcul folosind doar $\min(m, n)$ elemente şi un spaţiu suplimentar $O(1)$.

16.3-5 Daţi un algoritm de complexitate în timp $O(n^2)$ care găseşte cel mai lung subşir monoton crescător al unui şir de n numere.

16.3-6 $\star$ Scrieţi un algoritm cu complexitate în timp $O(n \lg n)$ care găseşte cel mai lung subşir monoton crescător al unui şir de n numere. (*Indicaţie:* observaţi că ultimul element al unui subşir candidat de lungime i este cel puţin la fel de mare ca ultimul element al subşirului candidat de lungime $i - 1$. Păstraţi subşirurile candidate legându-le prin secvenţa de intrare.)

16.4. Triangularea optimă a poligoanelor

În acest paragraf vom investiga problema determinării unei triangulări optime a unui poligon convex. Această problemă geometrică prezintă multe similarităţi cu problema parantezării produsului de matrice, chiar dacă aceste asemănări nu sunt evidente.

Un ***poligon*** este o curbă închisă, liniară pe porţiuni, în plan. Mai explicit, este o curbă având capătul iniţial identic cu cel final şi formată dintr-o secvenţă de segmente de dreaptă, numite ***laturi*** ale poligonului. Un punct care uneşte două feţe ale poligonului se numeşte ***vârf***. Vom presupune, în continuare, că poligonul este ***simplu***, adică nu se auto-intersectează. Mulţimea de puncte din plan cuprinse de un poligon simplu formează ***interiorul*** poligonului, mulţimea punctelor aparţinând efectiv poligonului formează ***frontiera***, iar mulţimea de puncte care înconjoară poligonul se numeşte ***exterior***. Un poligon simplu este numit ***convex*** dacă, dându-se oricare două puncte de pe frontieră sau din interiorul său, toate punctele aparţinând segmentului care uneşte aceste două puncte se află fie pe frontieră fie în interiorul poligonului.

Un poligon convex poate fi reprezentat prin precizarea vârfurilor, de exemplu în sensul acelor de ceasornic. Adică, dacă $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ este un poligon convex, atunci cele n laturi ale sale sunt $\overline{v_0v_1}, \overline{v_1v_2}, \ldots, \overline{v_{n-1}v_n}$, unde vom considera v_n ca fiind v_0. (În general, vom presupune, implicit, că toate operaţiile aritmetice asupra indicilor vârfurilor poligonului sunt efectuate modulo numărul de vârfuri.)

Dându-se două vârfuri neadiacente v_i şi v_j, segmentul $\overline{v_iv_j}$ se numeşte ***coardă*** a poligonului. O coardă $\overline{v_iv_j}$ împarte poligonul în două poligoane: $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ şi $\langle v_j, v_{j+1}, \ldots, v_i \rangle$. O ***triangulare*** a poligonului este o mulţime T de coarde ale poligonului care îl împart în ***triunghiuri*** disjuncte (poligoane cu 3 laturi). Figura 16.4 prezintă două modalităţi de triangulare a unui poligon având 7 laturi. Într-o triangulare, coardele nu se intersectează (cu excepţia capetelor lor), iar mulţimea T este maximală, adică orice coardă care nu aparţine mulţimii T va intersecta o coardă din T. Laturile triunghiurilor produse de triangulare sunt fie coarde în triangulare, fie laturi ale poligonului. Fiecare triangulare a unui poligon convex cu n vârfuri are $n - 3$ coarde şi împarte poligonul în $n - 2$ triunghiuri.

Pentru problema ***triangulării optime a unui poligon***, se dau un poligon convex $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ şi o funcţie de ponderare w definită pe triunghiurile formate de laturile şi

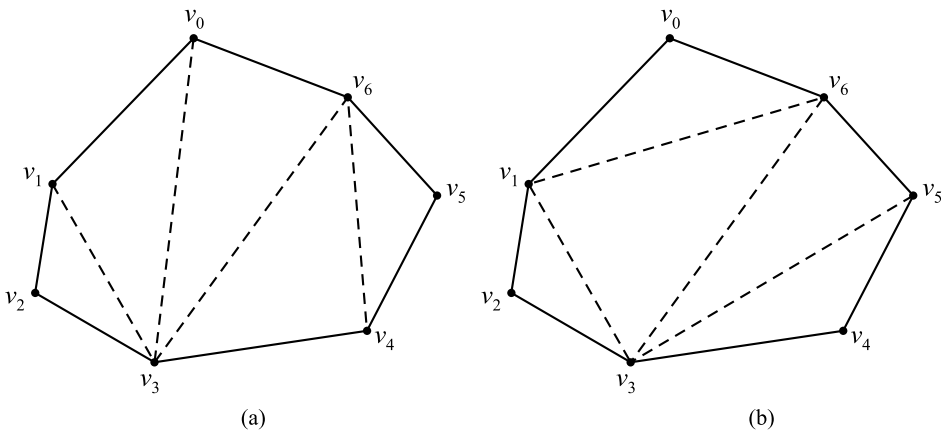


Figura 16.4 Două modalităţi de triangulare ale unui poligon convex. Fiecare triangulare a acestui poligon având 7 laturi are $7 - 3 = 4$ coarde şi împarte poligonul în $7 - 2 = 5$ triunghiuri.

coardele lui P. Problema care se pune este găsirea unei triangulări care minimizează suma ponderilor triunghiurilor din triangulare. O funcţie de ponderare care pare naturală este

$$w(\triangle v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|,$$

unde $|v_i v_j|$ reprezintă distanţa euclidiană de la v_i la v_j. Algoritmul pe care îl vom descrie este independent de alegerea funcţiei de ponderare.

Corespondenţa cu parantezarea

Există o surprinzătoare corespondenţă între triangularea unui poligon şi parantezarea unei expresii cum ar fi produsul unui şir de matrice. Această corespondenţă poate fi explicată cel mai bine folosind arbori.

O parantezare completă a unei expresii corespunde unui arbore binar complet, uneori numit şi ***arbore de analiză gramaticală*** a unei expresii. Figura 16.5(a) prezintă un arbore de analiză gramaticală pentru parantezarea produsului şirului de matrice

$$((A_1(A_2A_3))(A_4(A_5A_6))). \tag{16.6}$$

Fiecare frunză a arborelui de analiză gramaticală este etichetată cu unul dintre elementele atomice din expresie (în cazul de faţă matrice). Dacă rădăcina unui subarbore al arborelui de analiză are un subarbore stâng reprezentând o expresie E_s şi un subarbore drept reprezentând o expresie E_d, atunci subarborele însuşi reprezintă expresia $(E_s E_d)$. Aceasta este o corespondenţă biunivocă între arborii de analiză gramaticală şi expresiile complet parantezate având n elemente atomice.

O triangulare a unui poligon convex $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ poate fi reprezentată, de asemenea, printr-un arbore de analiză gramaticală. Figura 16.5(b) prezintă un astfel de arbore corespunzător triangulării poligonului din figura 16.4(a). Nodurile interne ale arborelui sunt coarde în triangulare, iar latura $\overline{v_0 v_6}$ este rădăcina. Frunzele sunt celelalte laturi ale poligonului. Rădăcina $\overline{v_0 v_6}$ este o latură a triunghiului $\triangle v_0 v_3 v_6$. Acest triunghi determină fiii rădăcinii: unul este

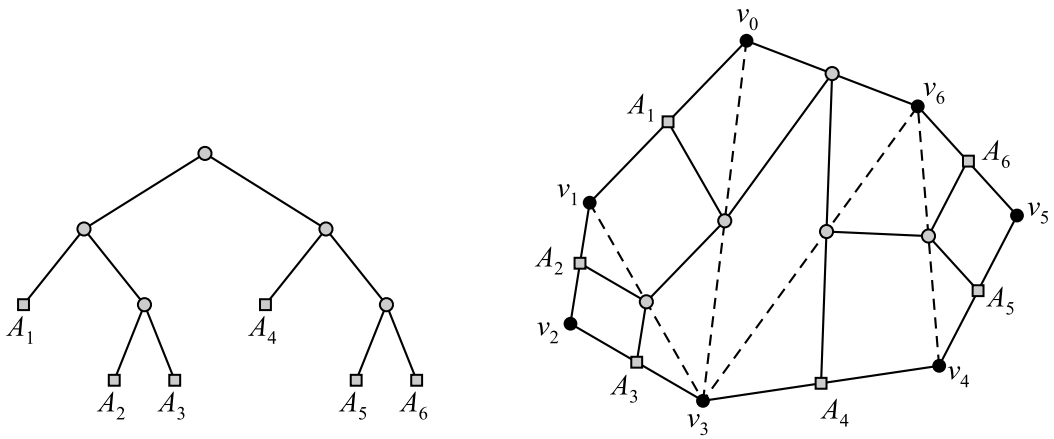


Figura 16.5 Arbori de analiză gramaticală. **(a)** Arbore de analiză pentru parantezarea produsului $((A_1(A_2A_3))(A_4(A_5A_6)))$ şi pentru triangularea poligonului având 7 laturi din figura 16.4(a). **(b)** Triangularea poligonului, suprapusă cu arborele de analiză. Fiecare matrice A_i corespunde laturii $\overline{v_{i-1}v_i}$ pentru $i = 1, 2, \ldots, 6$.

coarda $\overline{v_0v_3}$, iar celălalt este coarda $\overline{v_3v_6}$. Să observăm că acest triunghi împarte poligonul dat în trei părţi: triunghiul $\triangle v_0v_3v_6$ însuşi, poligonul $\langle v_0, v_1, \ldots, v_3\rangle$ şi poligonul $\langle v_3, v_4, \ldots, v_6\rangle$. Mai mult, cele două subpoligoane sunt formate, în întregime, din laturi ale poligonului considerat, cu excepţia rădăcinilor arborilor asociaţi lor, care sunt coardele $\overline{v_0v_3}$ şi $\overline{v_3v_6}$. Într-o manieră recursivă, poligonul $\langle v_0, v_1, \ldots, v_3\rangle$ conţine subarborele stâng al rădăcinii arborelui de analiză, iar poligonul $\langle v_3, v_4, \ldots, v_6\rangle$ conţine subarborele drept.

În general, o triangulare a unui poligon având n laturi corespunde unui arbore de analiză având $n-1$ frunze. Printr-un mecanism invers, se poate realiza o triangulare plecând de la un arbore de analiză. Astfel există o corespondenţă biunivocă între arborii de analiză şi triangulări.

Cum un produs de n matrice complet parantezat corespunde unui arbore de analiză având n frunze, el corespunde, de asemenea, unei triangulări a unui poligon având $(n+1)$ vârfuri. Figurile 16.5(a) şi (b) ilustrează această corespondenţă. Fiecare matrice A_i dintr-un produs $A_1A_2\cdots A_n$ corespunde unei laturi $\overline{v_{i-1}v_i}$ a unui poligon având $n+1$ vârfuri. O coardă $\overline{v_iv_j}$, unde $i < j$, corespunde unei matrice $A_{i+1..j}$ calculată în timpul evaluării produsului.

De fapt, problema înmulţirii unui şir de matrice este un caz special al problemei triangulării optime. Aceasta înseamnă că fiecare instanţă a produsului unui şir de matrice poate fi transformată într-o problemă de triangulare optimă. Dându-se un produs al unui şir de matrice $A_1A_2\cdots A_n$, vom defini un poligon convex cu $n+1$ vârfuri, $P = \langle v_0, v_1, \ldots, v_n\rangle$. Dacă matricea A_i are dimensiunile $p_{i-1} \times p_i$ pentru $i = 1, 2, \ldots, n$, vom defini funcţia de cost a triangulării astfel

$$w(\triangle v_iv_jv_k) = p_ip_jp_k.$$

O triangulare optimă a lui P relativă la această funcţie de cost produce un arbore de analiză pentru o parantezare optimă a produsului $A_1A_2\cdots A_n$.

Cu toate că reciproca nu este adevărată, adică problema triangulării optime *nu* este un

caz special al problemei înmulţirii unui şir de matrice, se poate arăta că procedura ORDINE-ŞIR-MATRICE din secţiunea 16.1, cu anumite modificări minore, rezolvă problema triangulării optime pentru un poligon având $n+1$ vârfuri. Trebuie doar să înlocuim secvenţa $\langle p_0, p_1, \ldots, p_n \rangle$ a dimensiunilor matricelor cu secvenţa $\langle v_0, v_1 \ldots, v_n \rangle$ a vârfurilor, să schimbăm toate referirile la p în referiri la v şi să schimbăm linia 9 astfel:

9: $q \leftarrow m[i,k] + m[k+1,j] + w(\triangle v_{i-1} v_k v_j)$

După execuţia algoritmului, elementul $m[1,n]$ conţine ponderea unei triangulări optime. În continuare vom justifica această afirmaţie.

Substructura unei triangulări optime

Să considerăm o triangulare optimă T a unui poligon având $n+1$ vârfuri $P = \langle v_0, v_1, \ldots, v_n \rangle$ care conţine triunghiul $\triangle v_0 v_k v_n$ pentru un anumit k cu $1 \leq k \leq n-1$. Ponderea asociată lui T este suma ponderilor triunghiului $\triangle v_0 v_k v_n$ şi a triunghiurilor corespunzătoare triangulării celor două subpoligoane $\langle v_0, v_1, \ldots, v_k \rangle$ şi $\langle v_k, v_{k+1}, \ldots, v_n \rangle$. Triangularea subpoligoanelor determinate de T trebuie, deci, să fie optimă, deoarece, dacă s-ar obţine pentru unul dintre ele o pondere de valoare mai mică, s-ar contrazice minimalitatea ponderii lui T.

O soluţie recursivă

Aşa cum, conform definiţiei, $m[i,j]$ este costul minim al calculului subprodusului şirului de matrice $A_i A_{i+1} \cdots A_j$, fie $t[i,j]$, pentru $1 \leq i < j \leq n$, ponderea triangulării optime a poligonului $\langle v_{i-1}, v_i, \ldots, v_j \rangle$. Prin convenţie, vom considera poligonul degenerat $\langle v_{i-1}, v_i \rangle$ ca având pondere 0. Ponderea unei triangulări optime a poligonului P este dată de $t[1,n]$.

Următorul pas este definirea recursivă a lui $t[i,j]$. Punctul de plecare este poligonul degenerat, cel cu două vârfuri: $t[i,i] = 0$ pentru $i = 1, 2, \ldots, n$. Când $j - i \geq 1$, poligonul $\langle v_{i-1}, v_i, \ldots, v_j \rangle$ are cel puţin trei vârfuri. Dorim să minimizăm suma dintre ponderea triunghiului $\triangle v_{i-1} v_k v_j$ şi ponderile triangulărilor optime pentru poligoanele $\langle v_{i-1}, v_i \ldots, v_k \rangle$ şi $\langle v_k, v_{k+1}, \ldots, v_j \rangle$, relativ la toate vârfurile v_k, pentru $k = i, i+1, \ldots, j-1$. În concluzie, funcţia recursivă este

$$t[i,j] = \begin{cases} 0 & \text{dacă } i = j, \\ \min\limits_{i \leq k \leq j-1} \{t[i,k] + t[k+1,j] + w(\triangle v_{i-1} v_k v_j)\} & \text{dacă } i < j. \end{cases} \tag{16.7}$$

Să comparăm această recurenţă cu (16.2), recurenţă pe care am construit-o pentru numărul $m[i,j]$, reprezentând numărul de înmulţiri scalare necesare pentru a calcula $A_i A_{i+1} \cdots A_j$. Cu excepţia funcţiei de ponderare, recurenţele sunt identice, aşadar, cu modificările minore menţionate mai sus, procedura ORDINE-ŞIR-MATRICE poate calcula ponderea unei triangulări optime. La fel ca ORDINE-ŞIR-MATRICE, procedura de triangulare optimă se execută într-un timp de ordinul $\Theta(n^3)$ şi foloseşte un spaţiu de memorie de ordinul $\Theta(n^2)$.

Exerciţii

16.4-1 Demonstraţi că orice triangulare a unui poligon convex având n vârfuri are $n-3$ coarde şi împarte poligonul în $n-2$ triunghiuri.

Figura 16.6 Şapte puncte în plan poziţionate pe o reţea unitară. **(a)** Cel mai scurt circuit închis, de lungime 24,88 .... Acest circuit nu este bitonic. **(b)** Circuitul bitonic cel mai scurt pentru aceeaşi mulţime de puncte. Lungimea sa este 25,58 ....

16.4-2 Profesorul Guinevere sugerează că un algoritm mai rapid pentru rezolvarea problemei triangulării optime poate exista în cazul particular în care ponderea triunghiului este chiar aria sa. Este această afirmaţie adevărată?

16.4-3 Să presupunem că funcţia de ponderare w este definită pe coardele triangulării şi nu pe triunghiuri. Ponderea unei triangulări relative la w este suma ponderilor coardelor triangulării. Arătaţi că problema triangulării optime cu coarde ponderate este doar un caz particular al problemei triangulării optime cu triunghiuri ponderate.

16.4-4 Determinaţi o triangulare optimă pentru un octogon regulat, având laturi egale cu unitatea. Utilizaţi funcţia de ponderare:

$$w(\triangle v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|,$$

unde $|v_i v_j|$ este distanţa euclidiană de la v_i la v_j. (Un poligon regulat este un poligon cu laturi şi unghiuri interne egale.)

Probleme

16-1 Problema euclidiană bitonică a comis-voiajorului

Problema euclidiană a comis-voiajorului este problema determinării celui mai scurt circuit care leagă o mulţime dată de n puncte în plan. Figura 16.6(a) prezintă soluţia unei probleme în care avem şapte puncte. În cazul general, problema este NP-completă, deci determinarea soluţiei nu se poate face în timp polinomial (a se vedea capitolul 36).

J. L. Bentley a sugerat că problema poate fi simplificată prin restricţionarea tipului de circuite căutate, şi anume la ***circuite bitonice***, adică circuite care încep din punctul cel mai din stânga, merg numai de la stânga la dreapta către punctul cel mai din dreapta, iar apoi merg numai către stânga, înapoi spre punctul de plecare. Figura 16.6(b) prezintă cel mai scurt circuit bitonic pentru cele şapte puncte. Determinarea unui astfel de circuit se poate face în timp polinomial.

Descrieţi un algoritm, având un timp de execuţie $O(n^2)$, pentru determinarea unui circuit bitonic optim. Se va presupune că două puncte distincte nu au aceeaşi abscisă. (*Indicaţie:* Parcugeţi de la stânga la dreapta, ţinând minte posibilităţile optime pentru cele două părţi ale circuitului.)

16-2 Tipărire uniformă

Să considerăm problema tipăririi uniforme a unui paragraf cu ajutorul unei imprimante. Textul de intrare este o succesiune de n cuvinte, de lungimi $l_1, l_2, \ldots, l_n$, lungimea fiind măsurată în caractere. Se doreşte tipărirea "uniformă" a paragrafului pe un anumit număr de linii, fiecare linie având cel mult M caractere. Acest criteriu de "uniformitate" funcţionează după cum urmează: dacă o linie conţine cuvintele de la cuvântul i la cuvântul j, cu $i \leq j$ iar între cuvinte se lasă exact un spaţiu, numărul de spaţii suplimentare de la sfârşitul unei linii este $M - j + i - \sum_{k=i}^{j} l_k$, număr care trebuie să fie nenegativ, astfel încât cuvintele să încapă pe linie. Dorim să minimizăm suma cuburilor numărului de spaţii suplimentare de la sfârşitul liniilor, în raport cu toate liniile, mai puţin ultima. Elaboraţi un algoritm folosind metoda programării dinamice, care să tipărească uniform la imprimantă un paragraf de n cuvinte. Analizaţi timpul de execuţie şi spaţiul de memorie necesare algoritmului.

16-3 Distanţa de editare

Există mai multe modalităţi prin care un terminal "inteligent" actualizează o linie dintr-un text, înlocuind un şir "sursă" $x[1..m]$ cu un şir "destinaţie" $y[1..n]$: un singur caracter al şirului sursă poate fi şters, înlocuit cu un alt caracter sau copiat în şirul destinaţie; sau două caractere adiacente ale şirului sursă pot fi interschimbate în timp ce sunt copiate în şirul destinaţie. După ce au fost realizate toate operaţiile, un întreg sufix al şirului sursă poate fi şters, o astfel de operaţie fiind cunoscută sub numele de "ştergere până la sfârşitul liniei" (sau, mai pe scurt, eliminare).

De exemplu, o modalitate de transformare a şirului sursă `algoritm` în şirul destinaţie `altruişti` utilizează următoarea secvenţă de operaţii.

Operaţie	Şir destinaţie	Şir sursă
copiază `a`	`a`	`lgoritm`
copiază `l`	`al`	`goritm`
înlocuieşte `g` cu `t`	`alt`	`oritm`
şterge `o`	`alt`	`ritm`
copiază `r`	`altr`	`itm`
inserează `u`	`altru`	`itm`
inserează `i`	`altrui`	`itm`
inserează `ş`	`altruiş`	`itm`
schimbă `it` în `ti`	`altruişti`	`m`
elimină `m`	`altruişti`	

Pentru a obţine acelaşi rezultat operaţiile pot fi executate în diverse ordini.

Fiecare dintre aceste operaţii: ştergerea, înlocuirea, copierea, inserarea, interschimbarea şi eliminarea, au asociate un cost. (Presupunem costul asociat înlocuirii unui caracter ca fiind mai mic decât costurile combinate ale ştergerii şi inserării; în caz contrar operaţia de înlocuire nu va fi utilizată). Costul unei secvenţe date de transformări este suma costurilor transformărilor individuale. Pentru secvenţa de mai sus, costul transformării cuvântului `algoritm` în `altruişti` este

$$(3 \cdot \text{cost(copiere)}) + \text{cost(înlocuire)} + \text{cost(ştergere)} + (4 \cdot \text{cost(inserare)}) + \text{cost(interschimbare)} + \text{cost(eliminare)}.$$

Dându-se două secvenţe $x[1..m]$ şi $y[1..n]$ şi o mulţime dată de costuri asociate operaţiilor, ***distanţa de editare*** de la x la y este costul celei mai puţin costisitoare secvenţe de transformări care îl converteşte pe x în y. Elaboraţi un algoritm, folosind metoda programării dinamice, pentru a determina distanţa de editare de la $x[1..m]$ la $y[1..n]$ şi pentru a afişa succesiunea optimă de transformări. Analizaţi timpul de execuţie şi spaţiul de memorie necesare.

16-4 Planificarea unei recepţii de către o companie

Profesorul McKenzie este consultantul preşedintelui Corporaţiei A.-B., care intenţionează să ofere o recepţie pentru companiile ce o compun. Corporaţia are o structură ierarhică, relaţiile de subordonare formează un arbore orientat, în care rădăcina este preşedintele. Fiecare angajat are ataşat un coeficient de "convieţuire", care este un număr real. Pentru a organiza o petrecere plăcută pentru toţi participanţii, preşedintele doreşte ca nici un angajat să nu se întâlnească cu şeful său direct.

a. Descrieţi un algoritm pentru alcătuirea listei invitaţilor. Scopul este de a maximiza suma coeficienţilor de convieţuire a invitaţilor. Analizaţi timpul de execuţie al algoritmului.

b. Cum poate profesorul să se asigure că preşedintele va fi invitat la propria sa petrecere?

16-5 Algoritmul Viterbi

Programarea dinamică poate fi utilizată în cazul unui graf orientat $G = (V, E)$ pentru recunoaşterea vorbirii. Fiecare muchie $(u, v) \in E$ este etichetată cu un sunet $\sigma(u, v)$ dintr-o mulţime finită Σ de sunete. Graful etichetat este un model formal al unei persoane care vorbeşte o limbă cu anumite restricţii. Fiecare drum în graf care începe într-un vârf dat $v_0 \in V$ corespunde unei secvenţe posibile de sunete produse de model. Eticheta unui drum este definită de concatenarea etichetelor muchiilor care compun drumul respectiv.

a. Fie un graf G cu muchiile etichetate, un vârf $v_0 \in V$ şi o secvenţă $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ de caractere din Σ. Descrieţi un algoritm eficient care returnează un drum din G, dacă există un astfel de drum, care începe din v_0 şi are eticheta s; în caz contrar, algoritmul va returna NU-EXISTĂ-DRUM. Analizaţi timpul de execuţie al algoritmului. (*Indicaţie:* pot fi utile noţiunile din capitolul 23.)

Să presupunem acum că fiecare muchie $(u, v) \in E$ are asociată, de asemenea, o probabilitate nenegativă $p(u, v)$ de traversare a muchiei (u, v) dinspre vârful u şi de producere a sunetului corespunzător. Suma probabilităţilor asociate muchiilor care pleacă din orice vârf este 1. Probabilitatea asociată unui drum se defineşte ca fiind produsul probabilităţilor asociate muchiilor care îl compun. Putem interpreta probabilitatea asociată unui drum care pleacă din v_0, ca fiind probabilitatea ca un "traseu aleator" care începe în v_0 să urmeze drumul specificat; selectarea muchiei care pleacă dintr-un vârf u se face probabilistic, conform probabilităţilor muchiilor care pleacă din u.

b. Extindeţi răspunsurile de la punctul (a) astfel încât, dacă este returnat un drum, acesta este *cel mai probabil drum* plecând din v_0 şi având eticheta s. Analizaţi timpul de execuţie al algoritmului.

Note bibliografice

R. Bellman a început studiul sistematic al metodei programării dinamice în 1955. Cuvântul "programare", ca şi în programarea liniară, face referire la utilizarea unei metode care furnizează soluţii în mod tabelar. Cu toate că tehnicile de optimizare care incorporează elemente de programare dinamică erau cunoscute anterior, Bellman a îmbogăţit domeniul cu o bază matematică solidă [21].

Hu şi Shing [106] au propus un algoritm având timp de execuţie $O(n \lg n)$ pentru problema înmulţirii şirului de matrice. Ei au demonstrat, de asemenea, corespondenţa între problema triangulării optime a poligoanelor şi problema înmulţirii unui şir de matrice.

Algoritmul având timp de execuţie $O(mn)$ pentru determinarea celui mai lung subşir comun pare a avea mai mulţi autori. Knuth [43] a pus întrebarea dacă există algoritmi având timp de execuţie mai redus decât timp pătratic pentru problema CMLSC. Masek şi Paterson [143] au răspuns afirmativ acestei întrebări propunând un algoritm cu timp de execuţie $O(mn/\lg n)$ unde $n \leq m$, iar secvenţele sunt extrase dintr-o mulţime mărginită. Pentru cazul particular în care nici un element nu apare mai mult decât o dată într-o secvenţă de intrare, Szymanski [184] a arătat că problema poate fi rezolvată într-un timp de ordinul $O((n+m) \lg (n+m))$. Multe dintre aceste rezultate se extind la problema calculării distanţelor de editare (problema 16-3).

17 Algoritmi greedy

Algoritmii aplicaţi problemelor de optimizare sunt, în general, compuşi dintr-o secvenţă de paşi, la fiecare pas existând mai multe alegeri posibile. Pentru multe probleme de optimizare, utilizarea metodei programării dinamice pentru determinarea celei mai bune soluţii se dovedeşte a fi o metodă prea complicată. Un ***algoritm greedy*** va alege la fiecare moment de timp soluţia ce pare a fi cea mai bună la momentul respectiv. Deci este vorba despre o alegere optimă, făcută local, cu speranţa că ea va conduce la un optim global. Acest capitol tratează probleme de optimizare ce pot fi rezolvate cu ajutorul algoritmilor greedy.

Algoritmii greedy conduc în multe cazuri la soluţii optime, dar nu întotdeauna... În secţiunea 17.1 vom prezenta mai întâi o problemă simplă dar netrivială, problema selectării activităţilor, a cărei soluţie poate fi calculată în mod eficient cu ajutorul unei metode de tip greedy. În secţiunea 17.2 se recapitulează câteva elemente de bază ale metodei greedy. În secţiunea 17.3 este descrisă o aplicaţie importantă a metodei greedy: proiectarea unor coduri pentru compactarea datelor şi anume codurile Huffman. În secţiunea 17.4 sunt investigate structuri combinatoriale numite "matroizi" pentru care algoritmii greedy produc întotdeauna soluţie optimă. În încheiere, în secţiunea 17.5 se exemplifică utilizarea matroizilor pentru problema planificării sarcinilor de timp unitar, sarcini având anumiţi termeni limită de realizare şi anumite penalizări în caz de neîndeplinire.

Metoda greedy este destul de puternică şi se aplică cu succes unui spectru larg de probleme. Capitolele următoare vor prezenta mai mulţi algoritmi ce pot fi priviţi ca aplicaţii ale metodei greedy, cum ar fi algoritmii de determinare a arborelui parţial de cost minim (capitolul 24), algoritmul lui Dijkstra pentru determinarea celor mai scurte drumuri pornind dintr-un vârf (capitolul 25) şi algoritmul lui Chvatal, o euristică pentru determinarea acoperirii unei mulţimi (capitolul 37). Arborii parţiali de cost minim sunt un exemplu clasic pentru metoda greedy. Cu toate că acest capitol şi capitolul 24 pot fi studiate independent, cititorul poate găsi utilă parcurgerea lor paralelă.

17.1. O problemă de selectare a activităţilor

Primul exemplu pe care îl vom considera este o problemă de repartizare a unei resurse mai multor activităţi care concurează pentru a obţine resursa respectivă. Vom vedea că un algoritm de tip greedy reprezintă o metodă simplă şi elegantă pentru selectarea unei mulţimi maximale de activităţi mutual compatibile.

Să presupunem că dispunem de o mulţime $S = 1, 2, \ldots, n$ de n ***activităţi*** care doresc să folosească o aceeaşi resursă (de exemplu o sală de lectură). Această resursă poate fi folosită de o singură activitate la un anumit moment de timp. Fiecare activitate i are un ***timp de pornire*** s_i şi ***un timp de oprire*** f_i, unde $s_i \leq f_i$. Dacă este selecţionată activitatea i, ea se desfăşoară pe durata intervalului $[s_i, f_i)$. Spunem că activităţile i şi j sunt ***compatibile*** dacă intervalele $[s_i, f_i)$ şi $[s_j, f_j)$ nu se intersectează (adică i şi j sunt compatibile dacă $s_i \geq f_j$ sau $s_j \geq f_i$). ***Problema selectării activităţilor*** constă din selectarea unei mulţimi maximale de activităţi mutual compatibile.

Un algoritm greedy pentru această problemă este descris de următoarea procedură, prezentată în pseudocod. Vom presupune că activităţile (adică datele de intrare) sunt ordonate crescător după timpul de terminare:

$$f_1 \leq f_2 \leq \ldots \leq f_n. \tag{17.1}$$

În caz contrar această ordonare poate fi făcută în timp $O(n \lg n)$. Algoritmul de mai jos presupune că datele de intrare s şi f sunt reprezentate ca vectori.

SELECTOR-ACTIVITĂŢI-GREEDY(s, f)
1: $n \leftarrow$ *lungime*$[s]$
2: $A \leftarrow \{1\}$
3: $j \leftarrow 1$
4: **pentru** $i \leftarrow 2, n$ **execută**
5: **dacă** $s_i \geq f_j$ **atunci**
6: $A \leftarrow A \cup \{i\}$
7: $j \leftarrow i$
8: **returnează** A

Operaţiile realizate de algoritm pot fi vizualizate în figura 17.1. În mulţimea A se introduc activităţile selectate. Variabila j identifică ultima activitate introdusă în A. Deoarece activităţile sunt considerate în ordinea nedescrescătoare a timpilor lor de terminare, f_j va reprezenta întotdeauna timpul maxim de terminare a oricărei activităţi din A. Aceasta înseamnă că

$$f_j = \max\{f_k : k \in A\} \tag{17.2}$$

În liniile 2–3 din algoritm se selectează activitatea 1, se iniţializează A astfel încât să nu conţină decât această activitate, iar variabila j ia ca valoare această activitate. În continuare liniile 4–7 consideră pe rând fiecare activitate i şi o adaugă mulţimii A dacă este compatibilă cu toate celelalte activităţi deja selectate. Pentru a vedea dacă activitatea i este compatibilă cu toate celelalte activităţi existente la momentul curent în A, este suficient, conform ecuaţiei (17.2), să fie îndeplinită condiţia din linia 5 adică momentul de pornire s_i să nu fie mai devreme decât momentul de oprire f_j al activităţii cel mai recent adăugate mulţimii A. Dacă activitatea i este compatibilă, atunci în liniile 6–7 ea este adăugată mulţimii A, iar variabila j este actualizată. Procedura SELECTOR-ACTIVITĂŢI-GREEDY este foarte eficientă. Ea poate planifica o mulţime S de n activităţi în $\Theta(n)$, presupunând că activităţile au fost deja ordonate după timpul lor de terminare. Activitatea aleasă de procedura SELECTOR-ACTIVITĂŢI-GREEDY este întotdeauna cea cu primul timp de terminare care poate fi planificată legal. Activitatea astfel selectată este o alegere "greedy" (lacomă) în sensul că, intuitiv, ea lasă posibilitatea celorlalte activităţi rămase pentru a fi planificate. Cu alte cuvinte, alegerea greedy maximizează cantitatea de timp neplanificată rămasă.

Demonstrarea corectitudinii algoritmului greedy

Algoritmii de tip greedy nu furnizează întotdeauna soluţiile optime. Cu toate acestea, algoritmul SELECTOR-ACTIVITĂŢI-GREEDY determină întotdeauna o soluţie optimă pentru o instanţă a problemei selectării activităţilor.

Teorema 17.1 Algoritmul SELECTOR-ACTIVITĂŢI-GREEDY furnizează soluţii de dimensiune maximă pentru problema selectării activităţilor.

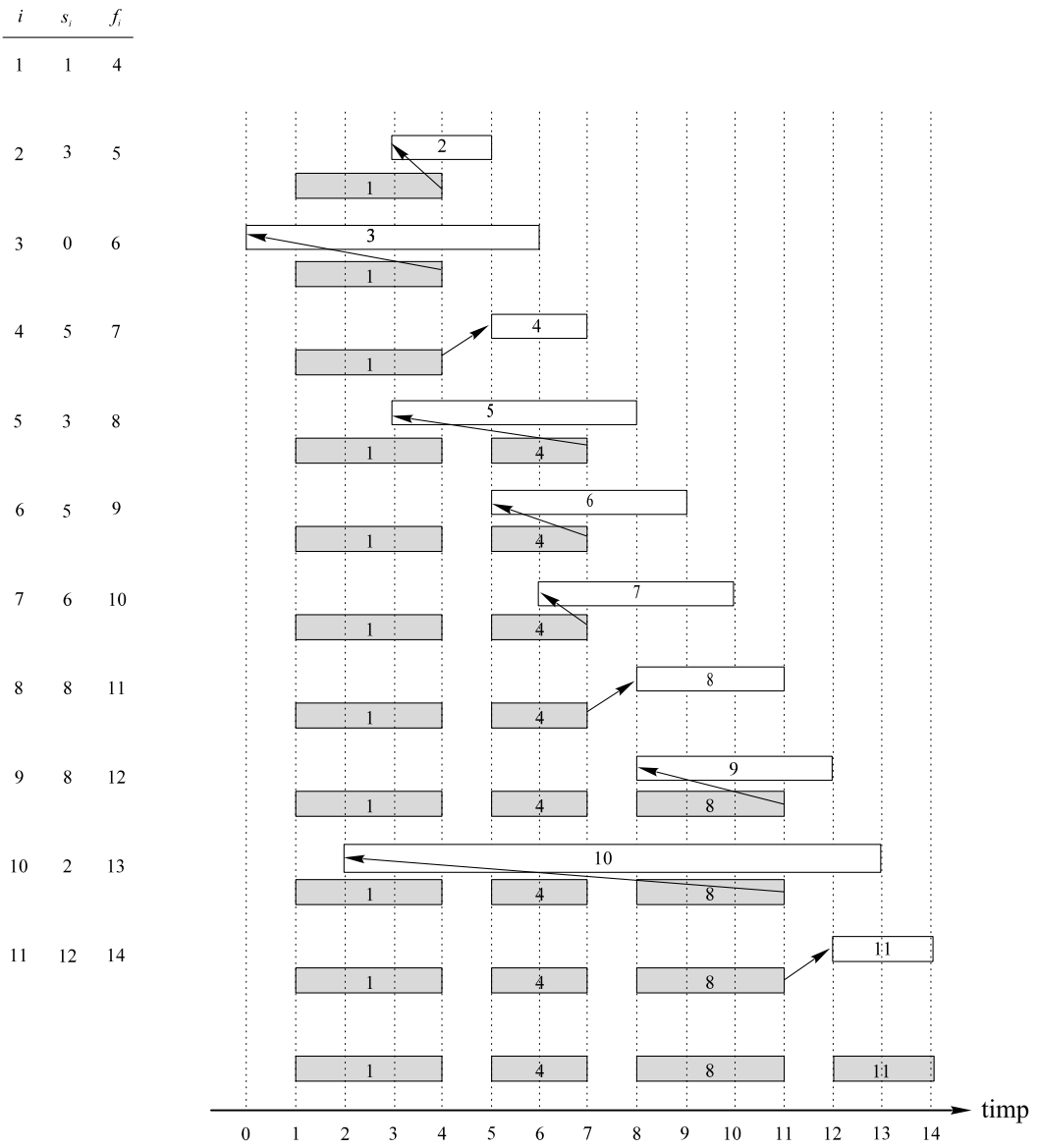


Figura 17.1 Operaţiile algoritmului SELECTOR-ACTIVITĂŢI-GREEDY asupra celor 11 activităţi date în stânga. Fiecare linie a figurii corespunde unei iteraţii din ciclul **pentru** din liniile 4–7. Activităţile care au fost selectate pentru a fi incluse în mulţimea A sunt haşurate, iar activitatea curentă i este nehaşurată. Dacă timpul de pornire s_i al activităţii i este mai mic decât timpul de terminare al activităţii j (săgeata dintre ele este spre stânga), activitatea este ignorată. În caz contrar (săgeata este îndreptată spre dreapta), activitatea este acceptată şi este adăugată mulţimii A.

Demonstraţie. Fie $S = \{1, 2, \ldots, n\}$ mulţimea activităţilor care trebuie planificate. Deoarece presupunem că activităţile sunt ordonate după timpul de terminare, activitatea 1 se va termina

cel mai devreme. Vrem să arătăm că există o soluţie optimă care începe cu activitatea 1, conform unei alegeri greedy.

Să presupunem că $A \subseteq S$ este o soluţie optimă pentru o instanţă dată a problemei selectării activităţilor. Vom ordona activităţile din A după timpul de terminare. Mai presupunem că prima activitate din A este activitatea k. Dacă $k = 1$, planificarea mulţimii A începe cu o alegere greedy. Dacă $k \neq 1$ vrem să arătăm că există o altă soluţie optimă B a lui S care începe conform alegerii greedy, cu activitatea 1. Fie $B = A - \{k\} \cup \{1\}$. Deoarece $f_1 \leq f_k$, activităţile din B sunt distincte, şi cum B are acelaşi număr de activităţi ca şi A, B este de asemenea optimă. Deci B este o soluţie optimă pentru S, care conţine alegerea greedy a activităţii 1. Am arătat astfel că există întotdeauna o planificare optimă care începe cu o alegere greedy.

Mai mult, o dată ce este făcută alegerea greedy a activităţii 1, problema se reduce la determinarea soluţiei optime pentru problema selectării acelor activităţi din S care sunt compatibile cu activitatea 1. Aceasta înseamnă că dacă A este o soluţie optimă pentru problema iniţială, atunci $A' = A - \{1\}$ este o soluţie optimă pentru problema selectării activităţilor $S' = \{i \in S : s_i \geq f_1\}$. De ce? Dacă am putea găsi o soluţie B' pentru S' cu mai multe activităţi decât A', adăugarea activităţii 1 la B' va conduce la o soluţie B a lui S cu mai multe activităţi decât A, contrazicându-se astfel optimalitatea mulţimii A. În acest mod, după ce este făcută fiecare alegere greedy, ceea ce rămâne este o problemă de optimizare de aceeaşi formă ca problema iniţială. Prin inducţie după numărul de alegeri făcute, se arată că realizând o alegere greedy la fiecare pas, se obţine o soluţie optimă. ∎

Exerciţii

17.1-1 Elaboraţi un algoritm conform metodei programării dinamice pentru problema selectării activităţilor, bazat pe calcularea valorilor m_i, iterativ pentru $i = 1, 2, \ldots, n$, unde m_i este dimensiunea celei mai mari mulţimi de activităţi compatibile dintre activităţile $\{1, 2, \ldots, i\}$. Se va presupune că datele de intrare au fost ordonate conform ecuaţiei (17.1). Comparaţi timpul de execuţie al algoritmului găsit cu cel al procedurii SELECTOR-ACTIVITĂŢI-GREEDY.

17.1-2 Să presupunem că avem o mulţime de activităţi de planificat pentru un număr mare de săli de lectură. Se doreşte planificarea tuturor activităţilor folosindu-se cât mai puţine săli de lectură posibil. Elaboraţi un algoritm greedy eficient pentru a determina ce activitate trebuie să utilizeze o sală de lectură şi care anume.

(Această problemă este cunoscută de asemenea sub numele de ***problema colorării grafurilor interval***. Putem crea un graf interval ale cărui vârfuri sunt activităţile date şi ale cărui muchii conectează activităţile incompatibile. Determinarea celui mai mic număr de culori necesare pentru a colora fiecare vârf, astfel încât două vârfuri adiacente să nu aibă aceeaşi culoare, este echivalentă cu determinarea numărului minim de săli de lectură necesare planificării tuturor activităţilor date.)

17.1-3 Nu orice algoritm de tip greedy aplicat problemei selctării activităţilor produce o mulţime maximală de activităţi compatibile. Daţi un exemplu care să arate că tehnica selectării activităţii cu timpul de executare cel mai scurt dintre activităţile compatibile cu cele deja selectate, nu este corectă. Se cere acelaşi lucru pentru încercarea de a selecta întotdeauna activitatea care se suprapune cu cele mai puţine dintre activităţile rămase.

17.2. Elemente ale strategiei greedy

Un algoritm greedy determină o soluţie optimă a unei probleme în urma unei succesiuni de alegeri. La fiecare moment de decizie din algoritm este aleasă opţiunea care pare a fi cea mai potrivită. Această strategie euristică nu produce întotdeauna soluţia optimă, dar există şi cazuri când aceasta este obţinută, ca în cazul problemei selectării activităţilor. În acest paragraf vom prezenta câteva proprietăţi generale ale metodei greedy.

Cum se poate decide dacă un algoritm greedy poate rezolva o problemă particulară de optimizare? În general nu există o modalitate de a stabili acest lucru, dar există două caracteristici pe care le au majoritatea problemelor care se rezolvă prin tehnici greedy: proprietatea de alegere greedy şi substructura optimă.

Proprietatea de alegere greedy

Prima caracteristică a unei probleme este aceea de a avea ***proprietatea alegerii greedy***, adică se poate ajunge la o soluţie optimă global, realizând alegeri (greedy) optime local. Aici intervine diferenţa dintre algoritmii greedy şi programarea dinamică. La rezolvarea unei metode prin metoda programării dinamice la fiecare pas al algoritmului se face câte o alegere dar ea depinde de soluţiile subproblemelor. Într-un algoritm greedy se realizează orice alegere care pare a fi cea mai bună la momentul respectiv, iar subproblema rezultată este rezolvată după ce alegerea este făcută. Alegerea realizată de un algoritm greedy poate depinde de alegerile făcute până în momentul respectiv, dar nu poate depinde de alegerile ulterioare sau de soluţiile subproblemelor. Astfel, spre deosebire de programarea dinamică prin care se rezolvă subproblemele în manieră "bottom-up", o strategie greedy de obicei progresează în manieră "top-down", realizând alegeri greedy succesive şi reducând iterativ dimensiunea problemei respective.

Desigur, trebuie să demonstrăm că o alegere greedy la fiecare pas conduce la o soluţie optimă global, şi aceasta este o problemă mai delicată. De obicei, ca în cazul teoremei 17.1, demonstraţia examinează o soluţie optimă global. Apoi se arată că soluţia poate fi modificată astfel încât la fiecare pas este realizată o alegere greedy, iar această alegere reduce problema la una similară dar de dimensiuni mai reduse. Se aplică apoi principiul inducţiei matematice pentru a arăta că o alegere greedy poate fi utilizată la fiecare pas. Faptul că o alegere greedy conduce la o problemă de dimensiuni mai mici reduce demonstaţia corectitudinii la demonstrarea faptului că o soluţie optimă trebuie să evidenţieze o substructură optimă.

Substructură optimă

O problemă evidenţiază o ***substrucutră optimă*** dacă o soluţie optimă a problemei conţine soluţii optime ale subproblemelor. Această proprietate este cheia pentru aplicarea programării dinamice sau a unui algoritm greedy. Ca exemplu al unei structuri optime, să ne reamintim demonstraţia teoremei 17.1, unde se arată că dacă o soluţie optimă A a problemei selectării activităţilor începe cu activitatea 1, atunci mulţimea activităţilor $A' = A - \{1\}$ este o soluţie optimă pentru problema selectării activităţilor $S' = \{i \in S : s_i \geq f_1\}$.

Comparaţie între metoda Geedy şi metoda programării dinamice

Deoarece proprietatea de substructură optimă este exploatată atât de metoda greedy cât şi de cea a programării dinamice, poate exista tentaţia de a genera o soluţie prin metoda programării dinamice, cu toate că un algoritm greedy ar fi suficient, sau invers, se poate crede că un algoritm greedy poate produce o soluţie optimă în timp ce corect ar trebui aplicată metoda programării dinamice. Pentru a ilustra diferenţele dintre cele două metode vom analiza în continuare două variante ale unei probleme clasice de optimizare.

Problema 0–1 a rucsacului se formulează după cum urmează. Un hoţ care jefuieşte un magazin găseşte n obiecte; obiectul i are valoarea v_i şi greutatea w_i, unde v_i şi w_i sunt numere întregi. El doreşte să ia o încărcătură cât mai valoroasă posibil, dar nu poate căra în sac o greutate totală mai mare decât W, unde W este şi el un număr întreg. Ce obiecte trebuie să ia? (Această problemă poartă numele de problema 0-1 a rucsacului pentru că un obiect fie este luat în întregime fie deloc; hoţul nu poate lua o parte din obiect sau un acelaşi obiect de mai multe ori).

În ***problema fracţionară a rucsacului*** hoţului îi este permis să ia şi părţi din obiecte. Un obiect din problema 0-1 a rucsacului poate fi gândit ca un lingou de aur, iar un obiect din problema fracţionară a rucsacului poate fi imaginat ca o anumită cantitate de praf de aur.

La ambele probleme se evidenţiază proprietatea de substructură optimă. Pentru problema 0–1, considerăm încărcătura cea mai valoroasă cea având greutatea cel mult W. Dacă din încărcătură se scoate obiectul j, ceea ce rămâne trebuie să fie o încărcătură având greutatea cel mult $W - w_j$ pe care hoţul o poate lua dintre cele $n - 1$ obiecte iniţiale, mai puţin j. Pentru problema fracţionară vom considera că dacă îndepărtăm un obiect j de greutate w din încărcătura optimă, ceea ce rămâne trebuie să fie cea mai valoroasă încărcătură având greutatea cel mult $W - w$ pe care hoţul o poate lua dintre cele $n - 1$ obiecte iniţiale, plus greutatea $w_j - w$ din obiectul j.

Cu toate că problemele sunt similare, problema fracţionară a rucsacului poate fi rezolvată printr-un algoritm de tip greedy, în timp ce pentru pentru problema 0–1 nu este corectă o astfel de rezolvare. Pentru a rezolva problema fracţionară calculăm mai întâi valoarea v_i/w_i (valoare per greutate) pentru fiecare obiect. Respectând o strategie greedy, hoţul începe prin a lua cât poate de mult din obiectul având valoarea per greutate maximă. În cazul în care obiectul respectiv a fost epuizat iar hoţul mai poate căra, el va lua cât de mult posibil din obiectul cu următoarea valoare per greutate, în ordine descrescătoare şi aşa mai departe, până nu mai poate încărca în rucsac. Astfel, prin ordonarea valorilor per greutate, algoritmul greedy se execută în $O(n \lg n)$ unităţi de timp. Lăsăm ca exerciţiu demonstrarea faptului că problema fracţionară a rucsacului are proprietatea alegerii greedy.

Pentru a arăta că pentru problema 0–1 a rucsacului nu se poate folosi o strategie greedy, vom considera instanţa problemei ilustată în figura 17.2(a). În această figură există trei obiecte, iar rucsacul are capacitatea 50 de unităţi. Primul obiect cântăreşte 10 şi valorează 60. Obiectul al doilea are greutatea 20 şi valoarea 100, iar al treilea cântăreşte 30 şi are o valoarea 120. Astfel, valoarea/greutate a primului obiect este 6 care este mai mare decât valoarea/greutate a celui de-al doilea obiect (2) sau a celui de-al treilea obiect (4). Conform strategiei greedy, la început va fi selectat obiectul 1. Cu toate acestea, după cum se poate vedea din analiza din figura 17.2(b) soluţia optimă alege obiectele 2 şi 3 şi lasă obiectul 1 deoparte. Nici una din cele două soluţii posibile care conţin obiectul 1 nu sunt optime.

Pentru problema fracţionară similară, strategia greedy consideră mai întâi obiectul 1 şi

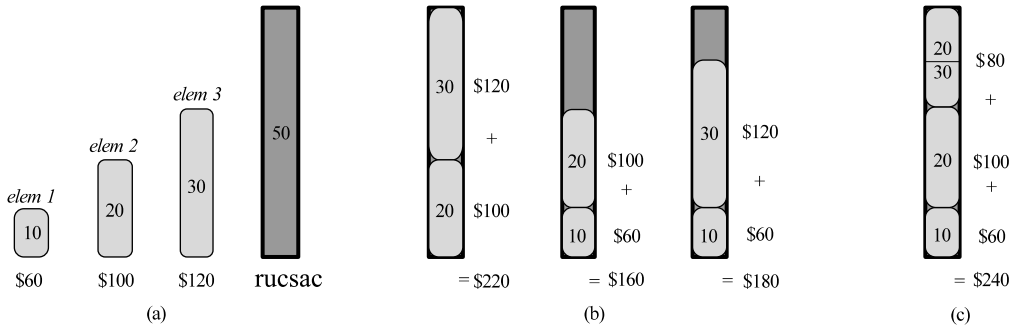


Figura 17.2 Strategia greedy nu funcţionează pentru problema 0–1 a rucsacului. **(a)** Hoţul trebuie să selecteze o submulţime a celor trei obiecte din figură a căror greutate nu trebuie să depăşească 50 de unităţi. **(b)** Submulţimea optimă cuprinde elementele 2 şi 3. Orice soluţie care conţine obiectul 1 nu este optimă, chiar dacă obiectul 1 are valoarea cea mai mare a câtului valoare/greutate. **(c)** Pentru problema fracţionară a rucsacului strategia considerării obiectelor în ordinea celei mai mari valori a câtului valoare/greutate conduce la soluţia optimă.

conduce la o soluţie optimă aşa cum se vede în figura 17.2(c).

Considerarea obiectului 1 nu este valabilă în problema rucsacului 0–1 deoarece hoţul nu poate să umple rucsacul până la capacitatea lui maximă, iar spaţiul liber micşorează câtul valoare/greutate a încărcăturii sale. În problema 0–1, atunci când luăm în considerare un obiect pentru a-l pune în rucsac, trebuie să comparăm soluţia subproblemei în care obiectul este inclus cu soluţia subproblemei în care obiectul lipseşte. Comparaţia trebuie realizată înainte de a face alegerea. Problema formulată în acest mod produce o serie de subprobleme care se suprapun – o caracteristică a programării dinamice, care poate fi într-adevăr folosită (vezi exerciţiul 17.2-2).

Exerciţii

17.2-1 Arătaţi că problema rucsacului fracţionar are propietatea alegerii greedy.

17.2-2 Rezolvaţi prin metoda programării dinamice problema 0-1 a rucsacului. Soluţia trebuie să fie calculată în $O(nW)$ unităţi de timp, unde n este numărul de obiecte, iar W este greutatea maximă a obiectelor pe care hoţul le poate încărca în rucsac.

17.2-3 Să presupunem că în problema 0–1 a rucsacului, ordinea obiectelor după sortarea crescătoare după greutate este aceeaşi ca şi după sortarea descrescătoare după valoare. Elaboraţi un algoritm eficient pentru a determina o soluţie optimă a acestei variante a problemei rucsacului şi argumentaţi corectitudinea algoritmului.

17.2-4 Profesorul Midas conduce un automobil de la Newark la Reno pe autostrada 80. Maşina sa are un rezervor de benzină care, dacă este plin, asigură călătoria pentru n kilometri. Harta pe care profesorul o are indică distanţa în kilometri între staţiile de benzină de pe drum. El doreşte să se oprească de cât mai puţine ori. Descrieţi o metodă eficientă prin care profesorul Midas poate să determine staţiile de benzină la care trebuie să se oprească şi arătaţi că strategia respectivă conduce la o soluţie optimă.

	a	b	c	d	e	f
Frecvenţă (în mii)	45	13	12	16	9	5
Cuvânt codificat; lungime fixă	000	001	010	011	100	101
Cuvânt codificat; lungime variabilă	0	101	100	111	1101	1100

Figura 17.3 O problemă de codificare a caracterelor. Un fişier de 100.000 de caractere conţine doar caracterele `a-f`, cu frecvenţele indicate. Dacă fiecărui caracter îi este asociat un cuvânt de cod pe 3 biţi, fişierul codificat va avea 300.000 de biţi. Folosind codificarea cu lungime variabilă, fişierul codificat va ocupa doar 224.000 de biţi.

17.2-5 Fie dată o mulţime $\{x_1, x_2, \ldots, x_n\}$ de puncte de pe dreapta reală. Descrieţi un algoritm eficient care determină cea mai mică mulţime de intervale închise de lungime egală cu unitatea, care conţin toate punctele date. Argumentaţi corectitudinea algoritmului.

17.2-6 $\star$ Arătaţi cum se poate rezolva problema fracţionară a rucsacului în $O(n)$ unităţi de timp. Presupuneţi că aveţi deja o soluţie pentru problema 10-2.

17.3. Coduri Huffman

Codurile Huffman reprezintă o tehnică foarte utilizată şi eficientă pentru compactarea datelor; în funcţie de caracteristicile fişierului care trebuie comprimat, spaţiul economisit este între 20% şi 90%. Algoritmul greedy pentru realizarea acestei codificări utilizează un tabel cu frecvenţele de apariţie ale fiecărui caracter. Ideea este de a utiliza o modalitate optimă pentru reprezentarea fiecărui caracter sub forma unui şir binar.

Să presupunem că avem un fişier ce conţine 100.000 de caractere, pe care dorim să îl memorăm într-o formă compactată. Frecvenţele de apariţie ale caracterelor în text sunt date de figura 17.3: există doar şase caractere diferite şi fiecare dintre ele apare de 45.000 de ori.

Există mai multe modalităţi de reprezentare a unui astfel de fişier. Vom considera problema proiectării unui ***cod binar al caracterelor*** (pe scurt ***cod***) astfel încât fiecare caracter este reprezentat printr-un şir binar unic. Dacă utilizăm un ***cod de lungime fixă***, avem nevoie de 3 biţi pentru a reprezenta şase caractere: `a`=000, `b`=001, ..., `f`=101. Această metodă necesită 300.000 de biţi pentru a codifica tot fişierul. Se pune problema dacă se poate face o compactare şi mai bună.

O ***codificare cu lungime variabilă*** poate îmbunătăţi semnificativ performanţele, atribuind caracterelor cu frecvenţe mai mari cuvinte de cod mai scurte iar celor cu frecvenţe mai reduse cuvinte de cod mai lungi. Figura 17.3 prezintă o astfel de codificare; şirul 0 având lungimea de 1 bit reprezintă caracterul `a` în timp ce şirul 1100 de lungime 4 reprezintă caracterul `f`. Această codificare necesită $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1.000 = 224.000$ biţi pentru a reprezenta un fişier şi economiseşte aproximativ 25% din spaţiu. Vom vedea că aceasta este de fapt o codificare-caracter optimă pentru acest fişier.

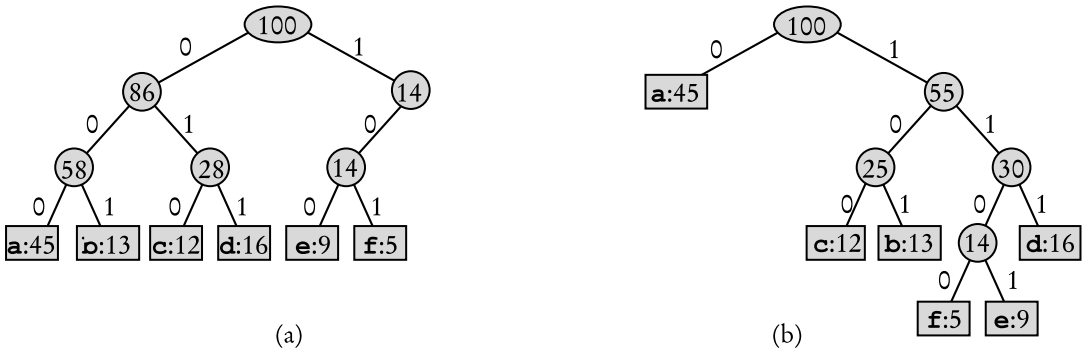


Figura 17.4 Arborii corespunzători schemelor de codificare din figura 17.3. Fiecare frunză este etichetată cu un caracter şi cu frecvenţa de apariţie a acestuia. Fiecare nod intern este etichetat cu suma ponderilor frunzelor din subarborele aferent. (**a**) Arborele corespunzător codificării de lungime fixă `a`=000, ..., `f`=101 (**b**) Arborele asociat codificării prefix optime `a`=0, `b`=101,..., `f`=1100.

Coduri prefix

Vom considera în continuare doar codificările în care nici un cuvânt de cod nu este prefixul altui cuvânt. Astfel de codificări se numesc ***codificări prefix***[1]. Se poate arăta (cu toate că nu vom proceda astfel în cazul de faţă) că o compresie optimă a datelor, realizată prin codificarea caracterelor, poate fi realizată şi prin codificarea prefix, deci considerarea codificării prefix nu scade din generalitate.

Codificarea prefix este utilă deoarece simplifică atât codificarea (deci compactarea) cât şi decodificarea. Codificarea este întotdauna simplă pentru orice codificare binară a caracterelor; se concatenează cuvintele de cod reprezentând fiecare caracter al fişierului. De exemplu, prin codificarea prefix având lungime variabilă din figura 17.3, putem codifica un fişier de 3 caractere `abc` astfel: 0·101·100=0101100, unde semnul "·" reprezintă operaţia de concatenare.

Decodificarea este de asemenea relativ simplă pentru codurile prefix. Cum nici un cuvânt de cod nu este prefixul altuia, începutul oricărui fişier codificat nu este ambiguu. Putem deci să identificăm cuvântul iniţial de cod, să îl "traducem" în caracterul original, să-l îndepărtăm din fişierul codificat şi să repetăm procesul pentru fişierul codificat rămas. În exemplul nostru, şirul 001011101 se translatează automat în 0·0·101·1101, secvenţă care se decodifică în `aabe`.

Procesul de decodificare necesită o reprezentare convenabilă a codificării prefix astfel încât cuvântul iniţial de cod să poată fi uşor identificat. O astfel de reprezentare poate fi dată de un arbore binar ale cărui frunze sunt caracterele date. Interpretăm un cuvânt de cod binar pentru un caracter ca fiind drumul de la rădăcină până la caracterul respectiv, unde 0 reprezintă "mergi la fiul stâng" iar 1 "mergi la fiul drept". În figura 17.4 sunt prezentaţi arborii pentru cele două codificări ale exemplului nostru. Observaţi că aceştia nu sunt arbori binari de căutare, deoarece frunzele nu trebuie să fie ordonate, iar nodurile interne nu conţin chei pentru caractere.

O codificare optimă pentru un fişier este întotdeuna reprezentată printr-un arbore binar *complet*, în care fiecare vârf care nu este frunză are doi fii (vezi exerciţiul 17.3-1).

Codificarea cu lungime fixă din exemplul de mai sus nu este optimă deoarece arborele asociat,

[1] "Codificare independentă de prefix" este probabil o denumire mai potrivită, dar termenul de "codificare prefix" este cel cunoscut în literatura de specialitate.

prezentat în figura 17.4(a), nu este un arbore binar complet: există două cuvinte de cod care încep cu 10..., dar nici unul care să înceapă cu 11.... Conform celor de mai sus ne putem restrânge atenţia numai asupra arborilor binari compleţi, deci dacă C este alfabetul din care fac parte caracterele, atunci arborele pentru o codificare prefix optimă are exact $|C|$ frunze, una pentru fiecare literă din alfabet, şi exact $|C| - 1$ noduri interne.

Dându-se un arbore T, corespunzător unei codificări prefix, este foarte simplu să calculăm numărul de biţi necesari pentru a codifica un fişier. Pentru fiecare caracter c din alfabet, fie $f(c)$ frecvenţa lui c în fişier şi să notăm cu $d_T(c)$ adâncimea frunzei în arbore (adică nivelul pe care se află). Să observăm că $d_T(c)$ reprezintă de asemenea cuvântul de cod pentru caracterul c. Numărul de biţi necesari pentru a codifica un fişier este

$$B(T) = \sum_{c \in C} f(c) d_T(c), \tag{17.3}$$

Vom numi acest număr ***costul*** arborelui T.

Construcţia unui cod Huffman

Huffman a inventat un algoritm greedy care construieşte o codificare prefix optimă numită ***codul Huffman***. Algoritmul construieşte arborele corespunzător codificării optime, într-o manieră "bottom-up". Se începe cu o mulţime de $| C |$ frunze şi se realizează o secvenţă de $| C | -1$ operaţii de "fuzionări" pentru a crea arborele final.

În algoritmul în pseudocod care urmează, vom presupune că C este o mulţime de n caractere şi fiecare caracter $c \in C$ este un obiect având o frecvenţă dată $f[c]$. Va fi folosită o coadă de priorităţi pentru a identifica cele două obiecte cu frecvenţa cea mai redusă care vor fuziona. Rezultatul fuzionării celor două obiecte este un nou obiect a cărui frecvenţă este suma frecvenţelor celor două obiecte care au fuzionat.

```
HUFFMAN(C)
1: n ← |C|
2: Q ← C
3: pentru i ← 1, n − 1 execută
4:     z ← ALOCĂ-NOD()
5:     x ← stânga[z] ← EXTRAGE-MIN(Q)
6:     y ← dreapta[z] ← EXTRAGE-MIN(Q)
7:     f[z] ← f[x] + f[y]
8:     INSEREAZĂ(Q, z)
9: returnează EXTRAGE-MIN(Q)
```

Pentru exemplul nostru, algoritmul lui Huffman lucrează ca în figura 17.5. Deoarece există 6 litere în alfabet, dimensiunea iniţială a cozii este $n = 6$, iar pentru a construi arborele sunt necesari 5 paşi de fuzionare. Arborele final reprezintă codificarea prefix optimă. Codificarea unei litere este secvenţa etichetelor nodurilor ce formează drumul de la rădăcină la literă.

În linia 2, coada de priorităţi Q este iniţializată cu caracterele din C. Ciclul **pentru** din liniile 3–8 extrage în mod repetat două noduri x şi y cu cea mai mică frecvenţă din coadă, şi le înlocuieşte în coadă cu un nou nod z, reprezentând fuziunea lor. Frecvenţa lui z este calculată în linia 7 ca fiind suma frecvenţelor lui x şi y. Nodul z are pe x ca fiu stâng şi pe y ca fiu drept. (Această ordine este arbitrară; interschimbarea fiilor stâng şi drept ai oricărui nod conduce la

o codificare diferită, dar de acelaşi cost.) După $n-1$ fuzionări, unicul nod rămas în coadă – rădăcina arborelui de codificare – este returnat în linia 9.

Analiza timpului de execuţie pentru algoritmul lui Huffman presupune implementarea lui Q sub forma unui heap binar (vezi capitolul 7). Pentru o mulţime C de n caractere, iniţializarea lui Q în linia 2 poate fi realizată în $O(n)$ unităţi de timp utilizând procedura CONSTRUIEŞTE-HEAP din secţiunea 7.3. Ciclul **pentru** din liniile 3–8 se execută de exact $|\ n\ | -1$ ori, şi cum fiecare operaţie asupra stivei necesită un timp $O(n \lg n)$, ciclul contribuie cu $O(n \lg n)$ la timpul de execuţie. Deci timpul total de execuţie a procedurii HUFFMAN pe o mulţime de n caractere este $O(n \lg n)$.

Corectitudinea algoritmului lui Huffman

Pentru a demonstra că algoritmul de tip greedy al lui Huffman este corect, vom arăta că problema determinării unei codificări prefix optime implică alegeri greedy şi are o substructură optimă. Următoarea lemă se referă la proprietatea alegerii greedy.

Lema 17.2 Fie C un alfabet în care fiecare caracter $c \in C$ are frecvenţa $f[c]$. Fie x şi y două caractere din C având cele mai mici frecvenţe. Atunci există o codificare prefix optimă pentru C în care cuvintele de cod pentru x şi y au aceeaşi lungime şi diferă doar pe ultimul bit.

Demonstraţie. Ideea demonstraţiei este de a lua arborele T reprezentând o codificare prefix optimă şi a-l modifica pentru a realiza un arbore reprezentând o altă codificare prefix optimă. În noul arbore, caracterele x şi y vor apare ca frunze cu acelaşi tată şi se vor afla pe nivelul maxim în arbore. Dacă putem realiza acest lucru, atunci cuvintele lor de cod vor avea aceeaşi lungime şi vor diferi doar pe ultimul bit.

Fie b şi c două caractere reprezentând noduri terminale (frunze) fraţi situate pe nivelul maxim al arborelui T. Fără a restrânge generalitatea, vom presupune că $f[b] \le f[c]$ şi $f[x] \le f[y]$. Cum $f[x]$ şi $f[y]$ sunt frunzele cu frecvenţele cele mai scăzute, în această ordine, iar $f[b]$ şi $f[c]$ sunt deci frecvenţe arbitrare, în ordine, avem $f[x] \le f[b]$ şi $f[y] \le f[c]$. După cum se vede în figura 17.6, vom schimba în T poziţiile lui b şi x pentru a produce arborele T', iar apoi vom schimba în T' poziţiile lui c şi y pentru a obţine arborele T''. Conform ecuaţiei (17.3), diferenţa costurilor lui T şi T' este

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f[x]d_T(x) + f[b]d_T(b) - f[x]d_{T'}(x) - f[b]d_{T'}(b) \\ &= f[x]d_T(x) + f[b]d_T(b) - f[x]d_{T'}(b) - f[b]d_{T'}(x) \\ &= (f[b] - f[x])(d_T(b) - d_T(x)) \ge 0, \end{aligned}$$

deoarece atât $f[b] - f[x]$ şi $d_T[b] - d_T[x]$ sunt nenegative. Mai precis, $f[b] - f[x]$ este nenegativ deoarce x este frunza având frecvenţa minimă iar $d_T[b] - d_T[x]$ este nenegativ pentru că b este o frunză aflată pe nivelul maxim în T. În mod analog, deoarece interschimbarea lui y cu c nu măreşte costul, diferenţa $B(T) \le B(T'')$ este nenegativă. Astfel $B(T) \le B(T'')$, ceea ce implică $B(T'') = B(T)$. Aşadar T'' este un arbore optim în care x şi y apar ca noduri terminale fraţi, şi se află pe nivelul maxim, ceea ce trebuia demonstrat. ∎

Din lema 17.2 se deduce faptul ca procesul de construcţie a unui arbore optim prin fuzionări poate, fără a restrânge generalitatea, să înceapă cu o alegere greedy a fuzionării acelor două

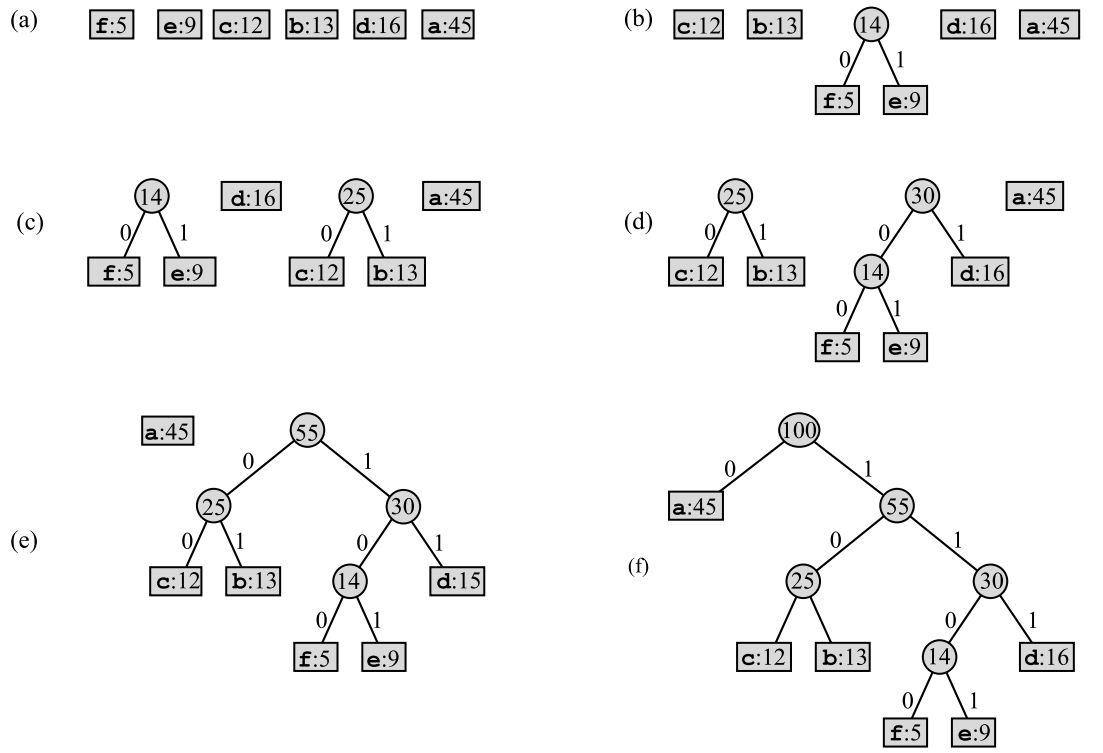


Figura 17.5 Paşii algoritmului Huffman pentru frecvenţele date în figura 17.3. Fiecare parte ilustrează conţinutul cozii ordonate crescător după frecvenţă. La fiecare pas, cei doi arbori cu cele mai scăzute frecvenţe fuzionează. Frunzele figurează ca dreptunghiuri ce conţin un caracter şi frecvenţa sa. Nodurile interne figurează ca cercuri ce conţin suma frecvenţelor fiilor lor. O muchie care leagă un nod intern cu fiii ei este etichetată cu 0 dacă este o muchie către un fiu stâng, respectiv cu 1 dacă este către fiul drept. Cuvântul de cod pentru o literă este secvenţa etichetelor de pe muchiile care leagă rădăcina de frunza asociată caracterului respectiv. **(a)** Mulţimea iniţială de $n = 6$ noduri, unul pentru fiecare literă. **(b)**–**(e)** Etape intermediare. **(f)** Arborele final.

caractere cu cea mai redusă frecvenţă. De ce este aceasta o alegere greedy? Putem interpreta costul unei singure fuzionări ca fiind suma frecvenţelor celor două obiecte care fuzionează. Exerciţiul 17.3-3 arată cum costul total al arborelui construit este suma costurilor obiectelor din care a fuzionat. La fiecare pas, dintre toate fuzionările posibile, algoritmul HUFFMAN o alege pe aceea care determină costul minim.

Următoarea lemă arată că problema construirii unei codificări prefix optime are proprietatea substructurii optime.

Lema 17.3 Fie T un arbore binar complet reprezentând o codificare prefix optimă peste un alfabet C, unde frecvenţa $f[c]$ este definită pentru fiecare caracter $c \in C$. Considerăm două caractere x şi y oarecare care apar ca noduri terminale fraţi în T, şi fie z tatăl lor. Atunci, considerând z ca un caracter având frecvenţa $f[z] = f[x] + f[y]$, arborele $T' = T - \{x, y\}$ reprezintă o codificare prefix optimă pentru alfabetul $C' = C - \{x, y\} \cup \{z\}$.

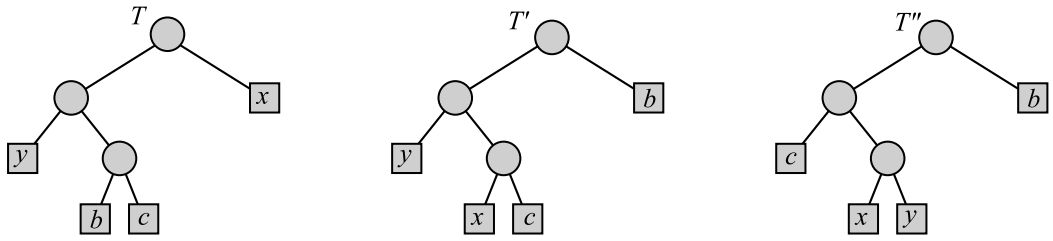


Figura 17.6 O ilustrare a pasului cheie din demonstaţia lemei 17.2. În arborele optim T, b şi c sunt două dintre frunzele aflate pe cel mai de jos nivel în arbore; aceste noduri se consideră fraţi. x şi y sunt două frunze pe care algoritmul Huffman le fuzionează primele; ele apar în poziţii arbitrare în T. Frunzele b şi x sunt interschimbate pentru a obţine arborele T'. Apoi, frunzele c şi y sunt interschimbate pentru a obţine arborele T''. Cum cele două interschimbări nu măresc costul, arborele rezultat T'' este de asemenea un arbore optim.

Demonstraţie. Vom arăta mai întâi că $B(T)$, costul arborelui T, poate fi exprimat în funcţie de costul $B(T')$ al arborelui T' considerând costurile componente din ecuaţia (17.3). Pentru fiecare $c \in C - \{x, y\}$, avem $d_T(c) = d_{T'}(c)$, şi $f[c]d_t(c) = f[c]d_{T'}(c)$. Cum $d_T(x) = d_T(y) = d_{T'}(z) + 1$, avem

$$f[x]d_T(x) + f[y]d_T(y) = (f[x] + f[y])(d_{T'}(z) + 1) = f[z]d_{T'}(z) + (f[x] + f[y]),$$

de unde deducem că

$$B(T) = B(T') + f[x] + f[y].$$

Dacă T' reprezintă o codificare prefix care nu este optimă pentru alfabetul C', atunci există un arbore T'' ale cărui frunze sunt caractere în C' astfel încât $B(T'') < B(T')$. Cum z este tratat ca un caracter în C', el va apare ca frunză în T''. Dacă adăugăm x şi y ca fiind fiii lui z în T'', atunci vom obţine o codificare prefix pentru C având costul $B(T'') + f[x] + f[y] < B(T)$, ceea ce intră în contradicţie cu optimalitatea lui T. Deci T' trebuie să fie optim pentru alfabetul C'. ∎

Teorema 17.4 Procedura HUFFMAN realizează o codificare prefix optimă.

Demonstraţie. Este imediată, din lemele 17.2 şi 17.3. ∎

Execiţii

17.3-1 Demonstraţi că un arbore binar care nu este complet nu poate corespunde unei codificări prefix optime.

17.3-2 Stabiliţi o codificare Huffman optimă pentru următoarea secvenţă de frecvenţe bazată pe primele 8 numere din şirul lui Fibonacci

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Puteţi generaliza răspunsul astfel încât să determinaţi codificarea optimă când frecvenţele sunt primele n numere din şirul lui Fibonacci?

17.3-3 Demonstraţi că, pentru o codificare, costul arborelui asociat poate fi calculat, de asemenea, ca sumă peste toate nodurile interne, ale frecvenţelor combinate ale celor doi fii ai nodului respectiv.

17.3-4 Arătaţi că pentru o codificare optimă, dacă sortăm caracterele în ordinea necrescătoare a frecvenţelor lor, lungimile cuvintelor de cod asociate sunt în ordine nedescrescătoare.

17.3-5 Fie $C = \{0, 1, \ldots, n-1\}$ o mulţime de caractere. Arătaţi că orice codificare prefix optimală pe C poate fi reprezentată sub forma unei secvenţe de $2n - 1 + n\lceil \lg n \rceil$ biţi. (*Indicaţie:* Pentru a specifica structura arborelui, utilizaţi $2n - 1$ biţi, conform parcurgerii acestuia.)

17.3-6 Generalizaţi algoritmul lui Huffman pentru cuvinte de cod ternare (adică acele cuvinte de cod care folosesc simbolurile 0, 1 şi 2) şi arătaţi că aceasta conduce la codificări ternare optime.

17.3-7 Fie dat un fişier de date conţinând un şir de caractere de 8 biţi, astfel încât toate cele 256 de caractere apar aproximativ cu aceeaşi frecvenţă: frecvenţa maximă este mai mică decât dublul frecvenţei minime. Arătaţi că, în acest caz, codificarea Huffman nu este mai eficientă decât utilizarea unui cod obişnuit având lungimea fixă de 8 biţi.

17.3-8 Arătaţi că de la nici o modalitate de comprimare nu se poate aştepta să compacteze un fişier care conţine caractere alese aleator, reprezentate pe 8 biţi, prin schimbarea unui singur caracter. (*Indicaţie:* Comparaţi numărul de fişiere cu numărul posibil de fişiere codificate).

17.4. Bazele teoretice ale metodei greedy

În această secţiune vom prezenta teoria care stă la baza algoritmilor greedy. Această teorie poate fi utilizată pentru a determina cazurile în care algoritmul greedy va conduce la soluţie optimă. Ea necesită cunoaşterea unor structuri combinatoriale numite "matroizi". Cu toate că această teorie nu acoperă toate cazurile pentru care se poate aplica o metodă greedy (de exemplu nu acoperă problema selectării activităţilor din secţiunea 17.1 sau problema codificării Huffman din secţiunea 17.3), ea cuprinde multe cazuri de interes practic. Mai mult, această teorie s-a dezvoltat într-un timp scurt şi a fost extinsă pentru a acoperi cât mai multe aplicaţii (a se vedea şi notele de la sfârşitul capitolului).

17.4.1. Matroizi

Un ***matroid*** este o pereche ordonată $M = (S, \mathcal{I})$ care satisface următoarele condiţii:

1. S este o mulţime finită nevidă.
2. $\mathcal{I}$ este o familie nevidă de submuţimi ale lui S, numite submulţimi ***independente*** ale lui S, astfel încât dacă $B \in \mathcal{I}$ şi $A \subseteq B$, atunci $A \in \mathcal{I}$. Spunem că $\mathcal{I}$ este ***ereditară*** dacă satisface această proprietate. Să observăm că mulţimea vidă $\emptyset$ este automat un element al lui $\mathcal{I}$.

3. Dacă $A \in \mathcal{I}$, $B \in \mathcal{I}$ şi $|A| < |B|$ atunci există un element $x \in B - a$ astfel încât $A \cup \{x\} \in \mathcal{I}$. Spunem că M satisface ***proprietatea de schimb***.

Cuvântul "matroid" a fost introdus de Hassel Whitney. El a studiat ***matroizii matriceali***, în care elementele lui S sunt liniile dintr-o matrice dată şi o mulţime de linii este independentă dacă liniile din mulţime sunt liniar independente în sensul uzual. Este uşor de arătat că această structură defineşte un matroid (vezi exerciţiul 17.4-2).

Un alt exemplu de matroid este următorul: considerăm ***matroidul grafic*** $M_G = (S_G, \mathcal{I}_G)$ definit în termenii unui graf neorientat $G = (V, E)$ după cum urmează:

- Mulţimea S_G este mulţimea E a muchiilor grafului G.
- Dacă A este o submulţime a lui E atunci $A \in \mathcal{I}_G$ dacă şi numai dacă A nu are cicluri. Aceasta înseamnă că o mulţime de muchii este independentă dacă şi numai dacă formează o pădure.

Matroidul grafic este strâns legat de problema arborelui de acoperire minimă care este prezentată în detaliu în capitolul 24.

Teorema 17.5 Dacă G este un graf neorientat, atunci $M_G = (S_G, \mathcal{I}_G)$ este un matroid.

Demonstraţie. Evident $S_G = E$ este mulţime finită. Mai mult, $\mathcal{I}_G$ este ereditară deoarece o submulţime a unei păduri este o pădure. Altfel spus, eliminarea unor muchii dintr-o mulţime de muchii ce nu formează cicluri nu poate crea cicluri noi. Astfel, rămâne de arătat că M_G satisface proprietatea de schimb. Să presupunem că A şi B sunt păduri ale lui G şi $|B| > |A|$. Cu alte cuvinte, A şi B sunt mulţimi aciclice de muchii şi B are mai multe muchii decât A.

Atunci, conform teoremei 5.2 o pădure având k muchii conţine exact $|V| - k$ arbori. (Pentru a arăta acest lucru în alt mod se pleacă cu $|V|$ arbori şi nici o muchie. Apoi fiecare muchie care este adăugată pădurii, reduce cu 1 numărul de arbori). Astfel, pădurea conţine $|V| - |A|$ arbori şi pădurea B conţine $|V| - |B|$ arbori.

Cum pădurea B are mai puţini arbori decât A, ea trebuie să conţină un arbore T cu vârfuri aflate în arbori diferiţi din pădurea A. Mai mult, dearece T este conex, el trebuie să conţină o muchie (u, v) astfel încât vârfurile u şi v să fie în arbori diferiţi din pădurea A. Cum muchia (u, v) leagă vârfuri în doi arbori diferiţi din pădurea A, ea poate fi adăugată pădurii A fără a forma un ciclu. Atunci, M_G satisface proprietatea de schimb, ceea ce încheie demonstraţia faptului că M_G este un matroid. ∎

Dându-se un matroid $M = (S, \mathcal{I})$, un element $x \notin A$ se numeşte ***extensie*** a lui $A \in \mathcal{I}$ dacă x poate fi adăugat la A cu păstrarea în acelaşi timp a independenţei; aceasta înseamnă că x este o extensie a lui A dacă $A \cup \{x\} \in \mathcal{I}$. Drept exemplu, să considerăm un matroid grafic M_G. Dacă A este o mulţime independentă de muchii, atunci muchia e este o extensie a lui A dacă şi numai dacă e nu este în A şi adăugarea ei la A nu creează un ciclu.

Dacă A este o submulţime independentă într-un matroid M, spunem că A este maximală dacă nu are extensii. Aceasta înseamnă că A este maximală dacă nu este inclusă în nici o submulţime independentă mai mare a lui M. Următoarea proprietate este utilizată frecvent.

Teorema 17.6 Toate submulţimile independente maximale dintr-un matroid au aceeaşi dimensiune.

Demonstraţie. Presupunem, prin reducere la absurd, că A este o submulţime independentă maximală a lui M şi există o altă submulţime B a lui M cu aceleaşi proprietăţi. Atunci, din proprietatea de schimb, rezultă că A este extensibilă la o mulţime mai mare $A \cup \{x\}$ pentru un $x \in B - A$, ceea ce contrazice presupunerea că A este maximală. ∎

Pentru a ilustra această teoremă, să construim matroidul grafic M_G pentru un graf conex neorientat G. Fiecare submulţime independentă maximală a lui M_G poate fi un arbore liber cu exact $|V| - 1$ muchii ce leagă toate vârfurile din G. Un astfel de arbore se numeşte ***arbore de acoperire*** al lui G.

Spunem că un matroid $M = (S, \mathcal{I})$ este ***ponderat*** dacă are asociată o funcţie de ponderare w care atribuie o pondere pozitivă $w(x)$ fiecărui element $x \in S$. Funcţia de ponderare w se extinde la submulţimile din S prin însumare:

$$w(A) = \sum_{x \in A} w(x)$$

pentru orice $A \subseteq S$. De exemplu, dacă $w(e)$ desemnează lungimea unei muchii e într-un matroid grafic M_G, atunci $w(A)$ este lungimea totală a muchiilor din A.

17.4.2. Algoritmi greedy pe un matroid ponderat

Numeroase probleme pentru care un algoritm greedy furnizează soluţii optime pot fi formulate astfel încât să conducă la determinarea unei submuţimi independente de pondere maximă într-un matroid ponderat. Cu alte cuvinte, se dă un matroid ponderat $M = (S, \mathcal{I})$ şi se doreşte determinarea unei mulţimi independente $A \in \mathcal{I}$ astfel încât $w(A)$ să fie maximizată. Vom spune că o astfel de submulţime, independentă şi de pondere maximă posibilă este submulţimea ***optimă*** a matroidului. Deoarece ponderea $w(x)$ a oricărui element x în S este pozitivă, o submulţime optimă este întotdeauna o submulţime maximală independentă. Este util ca A să fie cât mai mare cu putinţă.

De exemplu, în cazul ***arborelui de acoperire minim*** se consideră un graf neorientat, conex, $G = (V, E)$ şi o funcţie lungime w astfel încât $w(e)$ este lungimea (pozitivă) a muchiei e. (Vom folosi termenul de "lungime" pentru ponderi în matroidul asociat). Se cere determinarea unei submulţimi de muchii care uneşte toate vârfurile şi are lungime totală minimă. Pentru a privi aceasta ca o problemă de determinare a submulţimii optime a unui matroid, să considerăm matroidul M_G cu funcţia de ponderare w', unde $w'(e) = w_0 - w(e)$ şi w_0 este mai mare decât lungimea maximă a oricărei muchii. În acest matroid ponderat, toate ponderile sunt pozitive şi o submulţime optimă este un arbore de acoperire având lungimea minimă în graful original. Mai precis, fiecare submulţime independentă maximală A corespunde unui arbore de acoperire şi deoarece

$$w'(A) = (|V| - 1)w_0 - w(A)$$

pentru orice submulţime independentă maximală A, submulţimea independentă ce maximizează $w'(A)$ trebuie să minimizeze $w(A)$. În consecinţă, orice algoritm care poate găsi o submulţime optimă A într-un matroid arbitrar poate rezolva problema arborelui de acoperire minimă.

În capitolul 24 sunt prezentaţi algoritmi pentru problema arborelui de acoperire minimă, dar în acest paragraf vom descrie un algoritm greedy ce funcţionează pentru orice matroid ponderat. Algoritmul foloseşte ca date de intrare matroidul ponderat $M = (S, \mathcal{I})$ cu o funcţie asociată w

de pondere pozitivă w şi întoarce o submulţime optimă A. În algoritmul descris în pseudocod, vom nota componentele lui M cu $S[M]$ şi $\mathcal{I}[M]$, iar funcţia de ponderare cu w. Algoritmul este de tip greedy deoarece ia în considerare pe rând fiecare element $x \in S$ în ordinea necrescătoare a ponderilor şi îl adaugă imediat la mulţimea A deja obţinută dacă $A \cup \{x\}$ este independentă.

GREEDY(M, w)
1: $A \leftarrow \emptyset$
2: ordonează $S[M]$ necrescător după ponderile w
3: **pentru** fiecare $x \in S[M]$, în ordine necrescătoare după ponderea $w(x)$ **execută**
4: **dacă** $A \cup \{x\} \in \mathcal{I}(M)$ **atunci**
5: $A \leftarrow A \cup \{x\}$
6: **returnează** A

Elementele lui S sunt considerate pe rând, în ordinea necrescătoare a ponderilor. Elementul considerat x poate fi adăugat la A dacă menţine independenţa lui A. În caz contrar x este abandonat. Deoarece mulţimea vidă este independentă conform definiţiei unui matroid, şi cum x este adăugat la A doar dacă $A \cup \{x\}$ este independentă, prin inducţie, submulţimea A este întotdeauna independentă. Deci algoritmul GREEDY returnează întotdeauna o submulţime independentă A. Vom vedea în continuare că A este o submulţime de pondere maxim posibilă, deci că A este o submulţime optimă.

Timpul de execuţie al algoritmului GREEDY este uşor de calculat. Fie $n = |S|$. Etapa de sortare se realizează în $O(n \lg n)$.

Linia 4 este executată exact de n ori, o dată pentru fiecare element din S. Fiecare execuţie a liniei 4 necesită verificarea independenţei mulţimii $A \cup \{x\}$. Cum fiecare astfel de verificare se realizează în $O(f(n))$, întregul algoritm necesită $O(n \lg n + nf(n))$ unităţi de timp.

Vom arăta în cele ce urmează că algoritmul GREEDY returnează o submulţime optimă.

Lema 17.7 (Matroizii evidenţiază proprietatea alegerii greedy) Să presupunem că $M = (S, \mathcal{I})$ este un matroid ponderat cu o funcţie de ponderare w şi că S este ordonată necrescător după ponderi. Fie x primul element din S astfel încât $\{x\}$ este independentă, dacă un astfel de x există. Dacă x există, atunci există o submulţime optimă A a lui S care îl conţine pe x.

Demonstraţie. Dacă un astfel de x nu există atunci singura submulţime independentă este submulţimea vidă şi demonstraţia se încheie. Altfel, fie B o submulţime nevidă, optimă oarecare. Presupunem că $x \notin B$; altfel luăm $A = B$ şi demonstraţia se încheie.

Nici un element din B nu are ponderea mai mare decât $w(x)$. Să observăm că $y \in B$ implică $\{y\}$ este independentă deoarece $B \in \mathcal{I}$ şi $\mathcal{I}$ este ereditară. Alegerea făcută asupra lui x asigură $w(x) \geq w(y)$ pentru orice $y \in B$.

Mulţimea A se construieşte după cum urmează. Se pleacă cu $A = \{x\}$. Din alegerea lui x, A este independentă. Folosind proprietatea de schimb în mod repetat, se găseşte un nou element al lui B care poate fi adăugat la A până când $| A | = | B |$, astfel încât independenţa lui A este mereu păstrată. Atunci, $A = B - \{y\} \cup \{x\}$ pentru $y \in B$ şi deci

$$w(A) = w(B) - w(y) + w(x) \geq w(B)$$

Deoarece B este optimă, A trebuie să fie de asemenea optimă şi cum $x \in A$ lema este demonstrată. ∎

Lema 17.8 Fie $M = (S, \mathcal{I})$ un matroid oarecare. Dacă x este un element al lui S astfel încât x nu este o extensie a lui $\emptyset$, atunci x nu este o extensie a nici unei submulţimi independente A a lui S.

Demonstraţie. Demonstraţia se face prin reducere la absurd. Presupunem că x este o extensie a lui A dar nu a mulţimii vide $\emptyset$. Cum x este o extensie a lui A rezultă că $A \cup \{x\}$ este independentă. Deoarece $\mathcal{I}$ este ereditară, $\{x\}$ trebuie să fie independentă, ceea ce contrazice presupunerea că x nu este o extensie a lui $\emptyset$. ∎

Lema 17.8 arată că orice element care nu poate fi folosit la un anumit moment nu mai poate fi folosit niciodată ulterior. De aici se deduce că algoritmul GREEDY nu poate greşi, trecând peste elementele iniţiale din S care nu sunt extensii ale lui $\emptyset$, deoarece ele nu mai pot fi niciodată folosite.

Lema 17.9 (Matroizii pun în evidenţă proprietatea de substructură optimală) Fie x primul element ales de algoritmul GREEDY din S pentru matroidul ponderat $M = (S, \mathcal{I})$. Problema rămasă, de determinare a unei submulţimi independendente de pondere maximă care îl conţine pe x, se reduce la găsirea unei submulţimi independente de pondere maximă a matroidului ponderat $M' = (S', \mathcal{I}')$, unde

$$S' = \{y \in S : \{x, y\} \in \mathcal{I}\},\ \mathcal{I}' = \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\},$$

şi funcţia de ponderare pentru M' este funcţia de ponderare pentru M restricţionată la S'. (Numim M' ***contracţia*** lui M prin elementul x.)

Demonstraţie. Dacă A este o submulţime oarecare independentă de pondere maximă a lui M, care conţine elementul x, atunci $A' = A - \{x\}$ este o submulţime independentă a lui M'. Invers, orice submulţime independentă A' a lui M' conduce la o submulţime independentă $A = A' \cup \{x\}$ a lui M. Cum în ambele cazuri avem $w(A) = w(A') + w(x)$, o soluţie de pondere maximă în M, care îl conţine pe x, conduce la o soluţie de pondere maximă în M' şi invers. ∎

Teorema 17.10 (Corectitudinea algoritmului greedy pe matroizi) Dacă $M = (S, \mathcal{I})$ este un matroid ponderat cu funcţia de ponderare w, atunci apelul GREEDY(M, w) returnează o submulţime optimă.

Demonstraţie. Conform lemei 17.8 orice elemente care nu sunt considerate iniţial, deoarece nu sunt extensii ale mulţimii vide $\emptyset$, pot fi uitate definitiv, deoarece ele nu mai sunt folositoare. O dată ce primul element x este selectat, din lema 17.7 rezultă că algoritmul GREEDY nu poate greşi adăugând pe x la A pentru că există o submulţime optimă ce îl conţine pe x. În final lema 17.9 implică faptul că problema rămasă este una de determinare a unei submulţimi optime în matroidul M' care este contracţia lui M prin x. După ce procedura GREEDY atribuie lui A mulţimea x toţi ceilalţi paşi pot fi interpretaţi ca acţionând în matroidul $M' = (S', \mathcal{I}')$ deoarece B este independentă în M' dacă şi numai dacă $B \cup \{x\}$ este independentă în M, pentru orice mulţime $B \in \mathcal{I}'$. În consecinţă, operaţiile din algoritmul GREEDY vor determina o submulţime independentă de pondere maximă pentru M' şi toate operaţiile din algoritmul GREEDY vor găsi o submulţime independentă de pondere maximă pentru M. ∎

Exerciţii

17.4-1 Arătaţi că $(S, \mathcal{I}_k)$ este un matroid, unde S este o mulţime finită, iar $\mathcal{I}_k$ este mulţimea tuturor submulţimilor lui S de dimensiune cel mult k, unde $k \leq \mid S \mid$.

17.4-2 $\star$ Dându-se o matrice T de $n \times n$ numere reale, arătaţi că $(S, \mathcal{I})$ este un matroid, unde S este mulţimea coloanelor lui T şi $A \in \mathcal{I}$, dacă şi numai dacă în A coloanele sunt liniar independente.

17.4-3 $\star$ Arătaţi că dacă $(S, \mathcal{I})$ este un matroid, atunci şi $(S, \mathcal{I}')$ este un matroid, unde $\mathcal{I}' = \{A' : S - S' \text{ conţine mulţimi maximale } A \in \mathcal{I}\}$. Aceasta înseamnă că mulţimile independente maximale ale lui $(S', \mathcal{I}')$ sunt exact complementarele mulţimilor independente maximale ale lui $(S, \mathcal{I})$.

17.4-4 $\star$ Fie S o mulţime finită şi fie $S_1, S_2, \ldots, S_k$ o partiţie a lui S în submulţimi nevide disjuncte. Definim structura $(S, \mathcal{I})$ prin condiţia ca $\mathcal{I} = \{A :\mid A \cap S_i \mid \leq 1 \text{ pentru } i = 1, 2, \ldots, k\}$. Arătaţi că $(S, \mathcal{I})$ este un matroid. Aceasta înseamnă că matricea tuturor mulţimilor A, care conţine cel mult un element în fiecare bloc al partiţiei, determină o mulţime independentă de matroizi.

17.4-5 Arătaţi cum se transformă o funcţie de ponderare a unei probleme cu matroizi ponderaţi când soluţia optimă dorită este o submulţime independentă, maximală de pondere maximă, pentru a o aduce la o problemă standard de matroizi ponderaţi. Argumentaţi că transformarea este corectă.

17.5. O problemă de planificare a activităţilor

O problemă interesantă ce poate fi rezolvată folosind matroizi este problema planificării optime pe un singur procesor a unor activităţi care se execută într-o unitate de timp. Fiecare activitate are un termen de finalizare şi o penalizare care trebuie plătită dacă se depăşeşte termenul. Problema pare complicată dar poate fi rezolvată într-o manieră surprinzător de simplă, folosind un algoritm greedy.

O ***activitate într-o unitate de timp*** este o sarcină, de exemplu un program care se execută pe un calculator, care necesită exact o unitate de timp pentru a fi îndeplinită. Dându-se o submulţime finită S a unor astfel de activităţi, o ***planificare*** pentru S este o permutare a lui S ce specifică ordinea în care aceste sarcini trebuie realizate. Prima activitate din planificare începe la momentul 0 şi se termină la momentul 1, cea de-a doua începe la momentul 1 şi se termină la momentul 2 ş.a.m.d.

Problema ***planificării pe un singur procesor a activităţilor într-o unitate de timp cu termen de finalizare şi penalizări*** are următoarele date de intrare:

- o mulţime $S = \{1, 2, \ldots, n\}$ de n activităţi care se execută într-o unitate de timp;
- o mulţime de n întregi reprezentând ***termenele de finalizare*** $d_1, d_2, \ldots, d_n$ astfel încât $1 \leq d_i \leq n$ şi se presupune că activitatea i se termină la momentul d_i;

- o mulţime de n ponderi nenegative ***de penalizări*** $w_1, w_2, \ldots, w_n$ astfel încât o penalizare W_i este utilizată numai dacă activitatea i nu se termină la momentul d_i.

Se cere determinarea unei planificări pentru S care minimizează penalizarea totală pentru activitatea neterminată.

Considerăm o planificare dată. Spunem că o activitate este ***prematură*** în această planificare dacă se termină după termenul de finalizare. În caz contrar activitatea este ***în timp*** în planificare. O planificare arbitrară poate fi pusă întotdeauna într-o ***formă prematură***, în care fiecare activitate care se termină la timp precede activităţile întârziate. Să observăm că dacă o activitate prematură x, urmează unei activităţi întârziate y atunci poziţiile lui x şi y pot fi schimbate fără a afecta faptul că x este prematură şi y este întârziată.

Analog, afirmăm că o planificare arbitrară poate fi întotdeauna pusă sub ***forma canonică***, în care activităţile premature preced pe cele întârziate şi sunt planificate în ordine nedescrescătoare a termenelor de terminare. Pentru a realiza acest lucru se pune planificarea într-o primă formă prematură. Apoi, atâta timp cât există două activităţi în timp i şi j care se termină la momentele k şi $k+1$ în planificare cu termenele $d_j < d_i$, interschimbăm poziţia lui i cu cea a lui j. Cum activitatea j este prematură, înainte de interschimbare, $k+1 \leq d_j$. Atunci $k+1 < d_i$ şi astfel activitatea i este tot prematură după interschimbare. Activitatea j este mutată mai devreme în planificare, deci ea va fi de asemenea prematură după interschimbare.

Deci, căutarea unei planificări optime se reduce la determinarea unei mulţimi A de activităţi premature. O dată determinată A, putem crea planificarea curentă, listând întâi elementele din A în ordine nedescrescătoare a timpilor de terminare, apoi activităţile întârziate (adică $S - A$) în orice ordine, producând astfel o ordonare canonică a planificării optime.

Spunem că mulţimea A de activităţi este ***independentă*** dacă există o planificare pentru aceste activităţi astfel încât nici o activitate nu este întârziată. Evident, mulţimea activităţilor în timp pentru o planificare formează o mulţime independentă de activităţi. Fie $\mathcal{I}$ mulţimea tuturor mulţimilor independente de activităţi.

Considerăm problema determinării independenţei unei mulţimi date de activităţi A. Pentru $t = 1, 2, \ldots n$, fie $N_t(A)$ numărul de activităţi în A a căror termen de terminare este cel mult t.

Lema 17.11 Pentru orice mulţime de activităţi A, următoarele afirmaţii sunt echivalente:

1. A este mulţime independentă.
2. pentru $t = 1, 2, \ldots n$ avem $N_t(A) \leq t$.
3. dacă activităţile din A sunt planificate în ordinea nedescrescătoare a timpilor de terminare, atunci nici o activitate nu este întârziată.

Demonstraţie. Evident, dacă există t astfel încât $N_t(A) > t$, atunci nu există nici o modalitate de a face o planificare cu toate activităţile în timp pentru A, deoarece există mai mult de t activităţi care trebuie să se termine înainte de momentul t. Atunci (1) implică (2). Dacă (2) este adevărată atunci rezultă (3): nu există nici o modalitate de a ne bloca atunci când planificăm activităţile în ordinea nedescrescătoare a timpilor de terminare, deoarece din (2) rezultă că cel mai îndepărtat timp de terminare i este cel mult i. În sfârşit, evident, (3) implică (1). ∎

Folosind proprietatea 2 din lema 17.11 putem calcula cu uşurinţă dacă o mulţime dată de activităţi este independentă (vezi exerciţiul 17.5-2).

	Activitate						
	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Figura 17.7 O instanţă a problemei planificării activităţilor într-o unitate de timp cu termen de finalizare şi penalizări pe un singur procesor

Problema minimizării sumei de penalizare a activităţilor întârziate este aceeaşi cu problema maximizării sumei de penalizări ale activităţilor premature. Următoarea teoremă asigură corectitudinea utilizării algoritmului greedy pentru a determina o mulţime independentă A de activităţi cu penalizare totală maximă.

Teorema 17.12 Dacă S este o mulţime de activităţi într-o unitate de timp şi $\mathcal{I}$ este mulţimea tuturor mulţimilor independente de activităţi, atunci sistemul corespunzător $(S, \mathcal{I})$ este un matroid.

Demonstraţie. Fiecare submulţime a unei mulţimi independente de activităţi este evident independentă. Pentru a demonstra proprietatea de schimb să presupunem că B şi A sunt mulţimi independente de activităţi şi $\mid B \mid > \mid A \mid$. Fie k cel mai mare timp t astfel încât $N_t(B) \leq N_t(A)$. Cum $N_n(B) = \mid B \mid$ şi $N_n(A) = \mid A \mid$, dar $\mid B \mid > \mid A \mid$ trebuie să avem $k < n$ şi $N_j(B) > N_j(A)$ pentru orice j din intervalul $k + 1 \leq j \leq n$. Atunci B conţine mai multe activităţi cu timpul de terminare $k + 1$ decât A. Fie x o activitate în $B - A$ cu timpul de terminare $k + 1$. Fie $A' = A \cup \{x\}$.

Arătăm acum, folosind proprietatea 2 din lema 17.11, că A' trebuie să fie independentă. Pentru $1 \leq t \leq k$ avem $N_t(A') \leq N_t(B) \leq t$, deoarece B este independentă. Atunci A' este independentă ceea ce încheie demonstraţia faptului că $(S, \mathcal{I})$ este matroid. ∎

Conform teoremei 17.10 putem utiliza un algoritm greedy pentru a determina o mulţime independentă de pondere maximă de activităţi A. Putem atunci crea o planificare optimă având activităţile din A ca fiind activităţi premature. Această metodă este un algoritm eficient pentru planificarea activităţilor într-o unitate de timp cu termen de terminare şi penalizări pentru un singur procesor. Timpul de execuţie este $O(n^2)$ folosind algoritmul GREEDY deoarece fiecare din cele $O(n)$ verificări independente făcute de algoritm consumă $O(n)$ unităţi de timp (vezi exerciţiul 17.5-2). O implementare mai rapidă este dată în problema 17-3.

Figura 17.7 prezintă un exemplu pentru problema planificării activităţilor într-o unitate de timp cu termene de finalizare şi penalizări pe un singur procesor. În acest exemplu, algoritmul greedy selectează activităţile 1, 2, 3 şi 4, respinge activităţile 5 şi 6, iar în final acceptă activitatea 7. Planificarea optimă finală este

$$\langle 2, 4, 1, 3, 7, 5, 6 \rangle,$$

care are o penalizare totală egală cu $w_5 + w_6 = 50$.

Exerciţii

17.5-1 Rezolvaţi instanţa problemei planificării dată în figura 17.7 dar cu fiecare penalizare w_i înlocuită cu $80 - w_i$.

17.5-2 Arătaţi cum se utilizează proprietatea 2 din lema 17.1 pentru a determina în timpul $O(\mid A \mid)$ dacă o mulţime A de activităţi este independentă sau nu.

Probleme

17-1 Schimb de monede
Să considerăm problema schimbării a n cenţi utilizând cel mai mic număr de monede.

a. Descrieţi un algoritm greedy pentru a realiza schimbul în monede de 25, 50, 2 şi 1. Demonstraţi că algoritmul conduce la o soluţie optimă.

b. Să presupunem că monedele disponibile au valorile $c^0, c^1, \ldots, c^k$ pentru întregii $c > 1$ şi $k \geq 1$. Arătaţi că algoritmul greedy conduce mereu la soluţie optimă.

c. Daţi o mulţime de valori de monede pentru care algoritmul greedy nu conduce la o soluţie optimă.

17-2 Subgrafuri aciclice

a. Fie $G = (X, E)$ un graf neorientat. Folosind definiţia unui matroid, arătaţi că $(E, \mathcal{I})$ este un matroid, unde $A \in \mathcal{I}$ dacă şi numai dacă A este o submulţime aciclică a lui E.

b. ***Matricea de incidenţă*** a unui graf neorientat $G = (X, E)$ este o matrice M, de dimensiuni $\mid X \mid \times \mid E \mid$ astfel încât $M_{ve} = 1$ dacă muchia e este incidentă vârfului v şi $M_{ve} = 0$ altfel. Argumentaţi că o mulţime de coloane ale lui M este liniar indepenedentă dacă şi numai dacă mulţimea corespunzătoare de muchii este aciclică. Utilizaţi apoi rezultatul exerciţiului 17.4-2 pentru a furniza o altă demonstraţie a faptului că $(E, \mathcal{I})$ de la punctul a) este matroid.

c. Să presupunem că o pondere nenegativă $w(e)$ este asociată fiecărei muchii într-un graf neorientat $G = (X, E)$. Elaboraţi un algoritm eficient pentru a determina o submulţime aciclică a lui E de pondere totală maximă.

d. Fie $G = (V, E)$ un graf orientat oarecare şi fie $(E, \mathcal{I})$ definit astfel încât $A \in \mathcal{I}$ dacă şi numai dacă A nu conţine cicluri directe. Daţi un exemplu de graf orientat G astfel încât sistemul asociat $(E, \mathcal{I})$ nu este un matroid. Specificaţi ce condiţie din definiţia unui matroid nu este îndeplinită.

e. ***Matricea de incidenţă*** pentru un graf orientat $G = (V, E)$ este o matrice M de dimensiuni $\mid V \mid \times \mid E \mid$ astfel încât $M_{ve} = -1$ dacă muchia e pleacă din vârful v, $M_{ve} = 1$ dacă muchia e intră în vârful v şi $M_{ve} = 0$ altfel. Argumentaţi că dacă o mulţime de muchii ale lui G este liniar independentă atunci mulţimea corespunzătoare de muchii nu conţine cicluri orientate.

f. În exerciţiul 17.4-2 se afirmă că mulţimea mulţimilor liniar independente ale coloanelor oricărei matrice M formează un matroid. Explicaţi pe larg de ce rezultatele punctelor d) şi e) nu sunt în contradicţie. De ce nu poate exista o corespondenţă perfectă între o mulţime de muchii ca fiind aciclică şi mulţimea coloanelor corespunzătoare ale matricei de incidenţă ca fiind liniar independentă.

17-3 Planificarea variaţiilor

Să considerăm următorul algoritm pentru rezolvarea problemei din secţiunea 17.5 a planificării activităţilor într-o unitate de timp cu termene de finalizare şi penalizări. Fie n intervale de timp, iniţial vide, unde un interval de timp i este perioada de timp măsurată în unităţi ce se termină la momentul i. Considerăm activităţile în ordinea monoton descrescătoare a penalizărilor. Dacă la considerarea activităţii j există un interval de timp înainte sau cel târziu la termenul de finalizare d_j al lui j care este încă liber, se atribuie activitatea j celui mai din urmă astfel de interval, ocupându-l totodată. Dacă nu există un astfel de interval, activitatea j se atribuie celui mai din urmă interval neocupat.

a. Argumentaţi faptul că acest algoritm furnizează întotdeauna răspunsul corect.

b. Utilizaţi mulţimea disjunctă a pădurilor, prezentată în secţiunea 22.3 pentru a implementa eficient algoritmul. Presupuneţi că mulţimea activităţilor de intrare a fost ordonată monoton descrescător după penalizări. Analizaţi timpul de execuţie pentru această implementare.

Note bibliografice

Material bibliografic referitor la metoda greedy şi matroizi poate fi găsit în Lawler [132] şi Papadimitriou şi Steiglitz [154].

Algoritmii de tip greedy au apărut mai întâi în literatura referitoare la optimizarea combinatorială în 1971 într-un articol al lui Edmonds [62], cu toate că teoria matroizilor datează încă din 1935, ea fiind prezentată într-un articol al lui Whitney [200].

Demonstraţia corectitudinii algoritmului greedy pentru problema selectării activităţilor urmează ideea lui Gavril [80]. Problema planificării activităţilor este studiată în Lawler [132], Horowitz şi Sahni [105] şi Brassard şi Bratley [33].

Codificarea Huffman a fost inventată în 1952 [107]; lucrările lui Lelewer şi Hirschberg [136] despre tehnici de compactare a datelor sunt cunoscute din 1987.

O extensie a teoriei matroizilor la teoria greedoizilor a fost introdusă de Korte şi Lovász [127, 128, 129, 130]; ea generalizează teoria prezentată în acest capitol.

18 Analiza amortizată

Într-o ***analiză amortizată*** timpul necesar execuţiei unei secvenţe de operaţii asupra unei structuri de date este măsurat, în medie, pentru toate operaţiile efectuate. Analiza amortizată poate fi folosită pentru a arăta că, determinând media pentru o secvenţă de operaţii, costul unei operaţii este mic, chiar dacă o anume operaţie este costisitoare. Analiza amortizată diferă de analiza pentru cazul mediu prin faptul că nu este luată în calcul probabilitatea; o analiză amortizată garantează *performanţa medie a fiecărei operaţii pentru cazul cel mai defavorabil*.

Primele trei secţiuni din acest capitol prezintă cele mai uzuale trei tehnici folosite în analiza amortizată. Secţiunea 18.1 începe cu metoda de agregare în care se determină o margine superioară $T(n)$ a costului total al unei secvenţe de n operaţii. Costul amortizat pe operaţie va fi deci $T(n)/n$.

În secţiunea 18.2 se prezintă metoda de cotare, în care se determină costul amortizat al fiecărei operaţii. Dacă există mai multe tipuri de operaţii, fiecare astfel de tip poate avea un cost de amortizare distinct. Metoda de cotare supracotează unele operaţii la începutul secvenţei, memorând supracotarea ca pe un "credit plătit de la început" pentru anumite obiecte din structura de date. Acest credit este folosit ulterior pentru operaţii care sunt cotate sub costul lor real.

În secţiunea 18.3 se prezintă metoda de potenţial, asemănătoare cu metoda de cotare prin faptul că se determină costul amortizat pentru fiecare operaţie, iar la început operaţiile pot fi supracotate pentru a compensa subcotările ulterioare. Metoda de potenţial păstrează creditul drept "energie potenţială" a structurii de date, în loc să îl asocieze obiectelor individuale din structura de date.

Pentru ilustrarea celor trei metode vor fi prezentate două exemple. Primul se referă la o stivă cu operaţia suplimentară SCOATERE-MULTIPLĂ-DIN-STIVĂ, care poate extrage mai multe obiecte din stivă. Al doilea este un contor binar care numără începând de la 0 prin intermediul unicei operaţii INCREMENTEAZĂ.

Trebuie reţinut că, peste tot în acest capitol, cotările asociate în cadrul analizei amortizate sunt făcute doar pentru această analiză. Ele nu vor apărea în programe. Dacă, de exemplu, un credit este asociat obiectului x în cadrul analizei amortizate, aceasta nu înseamnă că în program va apare vreo variabilă $credit[x]$.

Aplicarea analizei amortizate pentru o structură de date particulară poate servi la optimizarea proiectării. Astfel, în secţiunea 18.4 vom folosi metoda de potenţial pentru analiza tabelelor care îşi măresc şi îşi micşorează dinamic dimensiunile.

18.1. Metoda de agregare

În ***metoda de agregare*** a analizei amortizate, vom arăta că, pentru orice n, o secvenţă de n operaţii necesită timpul $T(n)$ în *cazul cel mai defavorabil*. Drept urmare, în cazul cel mai defavorabil, costul mediu pe operaţie, numit şi ***cost amortizat***, va fi $T(n)/n$. Să remarcăm că acest cost amortizat se aplică fiecărei operaţii, chiar dacă în secvenţă apar mai multe tipuri de operaţii. Celelalte două metode studiate în acest capitol (metoda de cotare şi metoda de potenţial) pot asocia costuri de amortizare diferite la tipuri de operaţii diferite.

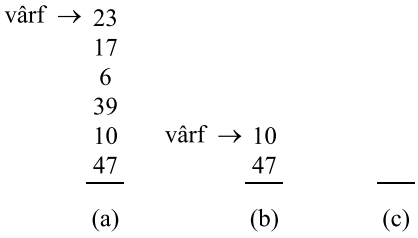


Figura 18.1 Modul de acţiune al lui SCOATERE-MULTIPLĂ-DIN-STIVĂ pe o stivă S, al cărei conţinut iniţial apare în (***a***). SCOATERE-MULTIPLĂ-DIN-STIVĂ$(S, 4)$ extrage 4 obiecte din vârful stivei, al cărei conţinut devine cel din (***b***). Următoarea operaţie este SCOATERE-MULTIPLĂ-DIN-STIVĂ$(S, 7)$, care conduce la stiva vidă - vezi (***c***) - deoarece în stivă rămăseseră mai puţin de 7 obiecte
.

Operatori de stivă

Ca prim exemplu pentru metoda de agregare, vom analiza stive pentru care se adaugă o nouă operaţie. În secţiunea 11.1 au fost introduşi cei doi operatori de stivă fundamentali, fiecare necesitând timpul $O(1)$:
PUNE-ÎN-STIVĂ(S, x) – introduce în stiva S obiectul x;
SCOATE-DIN-STIVĂ(s) – extrage un obiect din vârful stivei S şi îl returnează ca rezultat.
Cum fiecare dintre aceşti operatori necesită un timp $O(1)$, costul fiecăruia va fi considerat egal cu 1. De aceea, costul total al unei secvenţe de n operaţii PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ este n, iar timpul real pentru execuţia a n operatori este $\Theta(n)$.

Lucrurile devin mai interesante dacă adăugăm operatorul SCOATERE-MULTIPLĂ-DIN-STIVĂ(S, k) care extrage k obiecte din vârful stivei S; dacă stiva are mai puţin de k obiecte, este extras întreg conţinutul stivei. În următorul algoritm scris în pseudocod, operatorul STIVĂ-VIDĂ returnează ADEVĂRAT dacă stiva este vidă, respectiv FALS în caz contrar.

SCOATERE-MULTIPLĂ-DIN-STIVĂ(S, k)
1: **cât timp** nu STIVĂ-VIDĂ(S) şi $k \neq 0$ **execută**
2: SCOATE-DIN-STIVĂ(S)
3: $k \leftarrow k - 1$

În figura 18.1 este prezentat un exemplu de SCOATERE-MULTIPLĂ-DIN-STIVĂ.

Care este timpul de execuţie pentru operatorul SCOATERE-MULTIPLĂ-DIN-STIVĂ(S, k) pentru o stivă cu s obiecte? Timpul real de execuţie este liniar în numărul de operaţii SCOATE-DIN-STIVĂ efectiv executate, deci este suficient să analizăm operatorul SCOATERE-MULTIPLĂ-DIN-STIVĂ în funcţie de costul abstract 1 asociat fiecăruia dintre operatorii PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ. Numărul de iteraţii din ciclul ***pentru*** este numărul minim $\min(s, k)$ al obiectelor extrase din stivă. La fiecare iteraţie, în linia 2 are loc un apel SCOATE-DIN-STIVĂ. Ca urmare, costul total pentru SCOATERE-MULTIPLĂ-DIN-STIVĂ este $\min(s, k)$, iar timpul de execuţie real este o funcţie liniară în acest cost.

Să analizăm acum o secvenţă de n operatoţii PUNE-ÎN-STIVĂ, SCOATE-DIN-STIVĂ şi SCOATERE-MULTIPLĂ-DIN-STIVĂ pentru o stivă iniţial vidă. În cazul cel mai defavorabil, costul unui operator SCOATERE-MULTIPLĂ-DIN-STIVĂ din secvenţă este $O(n)$ deoarece mărimea stivei este cel mult n. De aceea, în cazul cel mai defavorabil, timpul este $O(n)$ şi, drept urmare,

costul unei secvenţe de n operatori este $O(n^2)$, deoarece pot exista un număr de $O(n)$ operatori SCOATERE-MULTIPLĂ-DIN-STIVĂ, fiecare având costul $O(n)$. Deşi această analiză este corectă, rezultatul $O(n^2)$ obţinut, considerând costul în cazul cel mai defavorabil pentru fiecare operator în parte, poate fi îmbunătăţit.

Prin utilizarea metodei de agregare a analizei amortizate, putem obţine o margine superioară mai bună prin considerarea întregii secvenţe de n operatori. Într-adevăr, deşi o operaţie SCOATERE-MULTIPLĂ-DIN-STIVĂ poate fi costisitoare, orice secvenţă de n operaţii PUNE-ÎN-STIVĂ, SCOATE-DIN-STIVĂ şi SCOATERE-MULTIPLĂ-DIN-STIVĂ plecând de la stiva vidă poate avea cel mult costul $O(n)$. De ce? Fiecare obiect poate fi extras cel mult o dată după introducerea sa. De aceea, numărul de apeluri SCOATE-DIN-STIVĂ, pentru stivă nevidă, inclusiv apelurile cuprinse în SCOATERE-MULTIPLĂ-DIN-STIVĂ, este cel mult egal cu numărul de operaţii PUNE-ÎN-STIVĂ, care este cel mult n. Pentru orice n, orice secvenţă de n operatori PUNE-ÎN-STIVĂ, SCOATE-DIN-STIVĂ şi SCOATERE-MULTIPLĂ-DIN-STIVĂ va avea costul total $O(n)$. Costul amortizat al unui operator va fi media corespunzătoare: $O(n)/n = O(1)$.

Precizăm din nou că, deşi timpul de execuţie şi costul mediu al unui operator de stivă este $O(1)$, nu s-a folosit un raţionament probabilist. Am determinat o margine $O(n)$ pentru *cazul cel mai defavorabil* pentru o secvenţă de n operatori. Împărţind acest cost total la n, am obţinut costul mediu pe operator, adică tocmai costul amortizat.

Un contor binar pentru incrementare

Pentru încă o exemplificare a metodei de agregare, vom considera un contor binar pe k biţi care îşi incrementează valoarea, plecând de la 0. Vom folosi un tablou de biţi $A[0..k-1]$ pentru acest contor; $lungime[A] = k$. Un număr binar x memorat în contor are cifra cea mai nesemnificativă memorată în $A[0]$, şi cea mai semnificativă memorată în $A[k-1]$, astfel încât $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Iniţial $x = 0$, deci $A[i] = 0$ pentru orice $i = 0, 1, \ldots, k-1$. Pentru a mări cu o unitate (modulo 2^k) valoarea contorului, folosim următoarea procedură:

INCREMENTEAZĂ(A)
1: $i \leftarrow 0$
2: **cât timp** $i < lungime[A]$ şi $A[i] = 1$ **execută**
3: $\quad A[i] \leftarrow 0$
4: $\quad i \leftarrow i + 1$
5: **dacă** $i < lungime[A]$ **atunci**
6: $\quad A[i] \leftarrow 1$

Acest algoritm este, în esenţă, cel implementat în hardware de un contor ripple-carry (vezi secţiunea 29.2.1). Figura 18.2 arată cum un contor binar este incrementat de 16 ori, plecând de la valoarea iniţială 0 şi terminând cu valoarea 16. La începutul fiecărei iteraţii a ciclului **cât timp**, în liniile 2–4 dorim să adunăm 1 pe poziţia i. Dacă $A[i] = 1$, atunci, prin adunarea lui 1, bitul de pe poziţia i devine 0 şi se obţine cifra de transport 1, care va fi adunată pe poziţia $i+1$ la următoarea iterare a ciclului. În caz contrar, ciclul se termină; dacă $i < k$, ştim că $A[i] = 0$, astfel încât adăugarea lui 1 pe poziţia i, comutând 0 în 1, se realizează pe linia 6. Costul fiecărei operaţii INCREMENTEAZĂ este liniar în numărul de biţi a căror valoare a fost comutată.

Asemănător exemplului cu stiva, o analiză brută produce o margine care este corectă, dar nu suficient de bună. O singură execuţie a lui INCREMENTEAZĂ are, în cazul cel mai defavorabil,

Valoarea contorului	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Costul total
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figura 18.2 Configuraţia unui contor binar pe 8 biţi a cărui valoare creşte de la 0 la 16 ca urmare a 16 operaţii INCREMENTEAZĂ. Biţii evidenţiaţi sunt cei care sunt afectaţi la obţinerea următoarei valori a contorului. Costul măsurat în numărul de biţi care îşi schimbă valoarea, este scris în dreapta. Este de remarcat că valoarea costului total nu depăşeşte de două ori numărul total de operaţii INCREMENTEAZĂ.

timpul $\Theta(k)$, corespunzător situaţiei în care toate componentele lui A sunt egale cu 1. Ca urmare, o secvenţă de n operaţii INCREMENTEAZĂ, plecând de la valoarea iniţială 0 a contorului, necesită în cazul cel mai defavorabil timpul $O(nk)$.

Putem îmbunătăţi analiza şi obţine costul $O(n)$ în cazul cel mai defavorabil pentru o secvenţă de n operaţii INCREMENTEAZĂ, observând că nu toţi biţii comută (îşi schimbă valoarea din 0 în 1 sau invers) la fiecare apel al procedurii INCREMENTEAZĂ. Aşa cum se arată în figura 18.2, $A[0]$ comută la fiecare apel al procedurii INCREMENTEAZĂ. Bitul $A[1]$ comută la fiecare două incrementări: o secvenţă de n operaţii INCREMENTEAZĂ, pentru un contor iniţial egal cu zero, face ca $A[1]$ să comute de $\lfloor n/2 \rfloor$ ori. Similar, bitul $A[2]$ comută la fiecare patru INCREMENTEAZĂări, deci de $\lfloor n/4 \rfloor$ ori. Mai general, pentru $i = 0, 1, \ldots, \lfloor \lg n \rfloor$, bitul $A[i]$ comută de $\lfloor n/2^i \rfloor$ ori într-o secvenţă de n operatori INCREMENTEAZĂ pentru un contor binar cu valoarea iniţială 0. Pentru $i > \lfloor \lg n \rfloor$, bitul $A[i]$ nu comută niciodată. Numărul de comutări pentru secvenţă este, ca urmare:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n,$$

conform ecuaţiei (3.4). Rezultă că, pentru o secvenţă de n operatori INCREMENTEAZĂ pentru un contor binar cu valoarea iniţială zero, timpul, în cazul cel mai defavorabil, este $O(n)$ şi, prin urmare, costul amortizat al fiecărei operaţii este $O(n)/n = O(1)$.

Exerciţii

18.1-1 Dacă am include un operator INSERARE-MULTIPLĂ-ÎN-STIVĂ printre operatorii de stivă, s-ar păstra marginea $O(1)$ pentru costul amortizat al operaţiilor pe stivă?

18.1-2 Arătaţi că, dacă s-ar include un operator DECREMENT în exemplul cu contorul binar pe k biţi, n operaţii ar putea necesita timpul $\Theta(nk)$.

18.1-3 Considerăm o secvenţă de n operaţii ce se execută pe o structură de date. Presupunem că operaţia i are costul i dacă i este o putere a lui 2, respectiv, are costul egal cu 1 în caz contrar. Aplicaţi o metodă de agregare pentru a determina costul de amortizare pe operaţie.

18.2. Metoda de cotare

În ***metoda de cotare*** a analizei amortizate, atribuim operaţiilor diferite, cotări diferite, unele unele operaţii fiind supracotate sau subcotate faţă de costul real. Valoarea cu care cotăm o operaţie se numeşte ***costul amortizat***. Când costul amortizat al unei operaţii depăşeşte costul real, diferenţa este atribuită la anumite obiecte din structura de date drept ***credit***. Acest credit poate fi folosit mai târziu pentru a ajuta să plătim pentru operaţiile al căror cost amortizat este mai mic decât costul lor real. Astfel, putem considera costul amortizat al unei operaţii ca fiind format din costul real şi creditul, care este fie depozitat, fie folosit. Aici apare deosebirea faţă de metoda de agregare, în care toate operaţiile au acelaşi cost amortizat.

Costurile amortizate ale operaţiilor trebuie alese cu grijă. Dacă dorim ca analiza cu costuri amortizate să evidenţieze că, în cel mai defavorabil caz, costul mediu pe operaţie este mic, costul amortizat total al unei secvenţe de operaţii trebuie să fie o margine superioară pentru costul real total al secvenţei. Mai mult, la fel ca în metoda de agregare, această relaţie trebuie să aibă loc pentru toate secvenţele de instrucţiuni. În acest mod, creditul total asociat structurii de date trebuie să fie, în permanenţă, nenegativ, deoarece reprezintă cantitatea cu care costurile amortizate totale depăşesc costurile reale totale. Dacă s-ar admite valori negative pentru creditul total (ca rezultat al subcotării iniţiale a operaţiilor, în speranţa de a returna acest aconto mai târziu), atunci costurile amortizate totale suportate la acel moment ar fi sub costurile totale reale suportate; pentru secvenţa de operaţii până la acel moment, costul total amortizat nu va constitui o margine superioară pentru costul real total. De aceea, trebuie să avem grijă ca în structura de date creditul să nu devină negativ.

Operaţii de stivă

Pentru a ilustra metoda de cotare a analizei amortizate, să ne întoarcem la exemplul referitor la stive. Reamintim costurile reale ale operaţiilor:

PUNE-ÎN-STIVĂ	1,
SCOATE-DIN-STIVĂ	1,
SCOATERE-MULTIPLĂ-DIN-STIVĂ	$\min(k, s)$,

unde k este argumentul furnizat la apelarea subprogramului SCOATERE-MULTIPLĂ-DIN-STIVĂ, iar s este mărimea stivei la apelare. Vom ataşa următoarele costuri amortizate:

PUNE-ÎN-STIVĂ	2,
SCOATE-DIN-STIVĂ	0,
SCOATERE-MULTIPLĂ-DIN-STIVĂ	0.

Să remarcăm că SCOATERE-MULTIPLĂ-DIN-STIVĂ are drept cost amortizat o constantă (0), în timp ce costul real este variabil. Aici toate cele trei costuri amortizate sunt de ordinul $O(1)$, deşi, în general, costurile amortizate ale operaţiilor considerate pot să difere asimptotic.

Vom arăta în continuare că vom putea "plăti" pentru orice secvenţă de operaţii de stivă prin cotarea costurilor amortizate. Presupunem că vom folosi o hârtie de 1\$ pentru fiecare unitate de cost. Pornim cu stiva vidă. Să ne reamintim de analogia din secţiunea 11.1 între structura de date stivă şi un vraf de farfurii. Când adăugăm o farfurie în stivă, plătim 1\$ pentru costul real al adăugării în stivă şi rămânem cu un credit de 1\$ (din cei 2\$ cotaţi) pe care îl punem pe farfurie. La orice moment de timp, pe fiecare farfurie din stivă apare creditul de 1\$.

Dolarul depus pe farfurie este o plată anticipată pentru costul extragerii ei din stivă. La execuţia unei operaţii SCOATE-DIN-STIVĂ, cotăm operaţia cu 0 şi plătim costul ei real folosind creditul depus pe stivă. Pentru a extrage o farfurie, luăm dolarul de pe acea farfurie şi îl folosim pentru a plăti costul real al operaţiei. Astfel, cotând un pic mai mult operaţia PUNE-ÎN-STIVĂ, nu trebuie să cotăm cu ceva operaţia SCOATE-DIN-STIVĂ.

Mai mult, nu trebuie să mai cotăm cu ceva operaţiile SCOATERE-MULTIPLĂ-DIN-STIVĂ. Pentru a extrage prima farfurie, luăm dolarul aflat pe ea şi îl folosim pentru a plăti costul real al operaţiei SCOATE-DIN-STIVĂ. Pentru a extrage a doua farfurie, avem din nou la dispoziţie un credit de 1\$ pe farfurie pentru a plăti operaţia SCOATE-DIN-STIVĂ şi aşa mai departe. În acest mod am cotat (plătit) suficient înainte de a executa operaţii SCOATERE-MULTIPLĂ-DIN-STIVĂ. Cu alte cuvinte, cum fiecare farfurie din stivă are pe ea un credit de 1\$ şi cum stiva conţine un număr nenegativ de farfurii, ne-am asigurat că valoarea creditului este întotdeauna nenegativă. Astfel, pentru *orice* secvenţă de n operaţii PUNE-ÎN-STIVĂ, SCOATE-DIN-STIVĂ şi SCOATERE-MULTIPLĂ-DIN-STIVĂ, costul total amortizat este o margine superioară a costului real total. Cum costul amortizat total este $O(n)$, costul real total are acelaşi ordin de mărime.

Incrementarea unui contor binar

Ca o a doua ilustrare a metodei de cotare, vom analiza acţiunea operaţiei INCREMENTEAZĂ asupra unui contor binar iniţializat cu zero. Aşa cum s-a arătat mai sus, timpul de execuţie al acestei operaţii este proporţional cu numărul de biţi ce comută; acest număr va fi folosit drept cost pentru acest exemplu. Vom folosi, din nou, o hârtie de 1\$ pentru a reprezenta fiecare unitate de cost (comutarea unui bit în cazul nostru).

Pentru analiza amortizată vom cota cu 2\$, drept cost amortizat, setarea unui bit pe 1. La setarea unui bit, vom folosi 1\$ (din cei 2\$) pentru a plăti setarea efectivă a bitului şi vom plasa celălalt dolar pe bit, drept credit. În fiecare moment, fiecărui 1 din contorul binar i s-a asociat un credit de 1\$, astfel încât nu trebuie plătit nimic pentru resetarea bitului pe 0: plătim resetarea cu dolarul aflat pe bit.

Putem determina acum costul amortizat al subprogramului INCREMENTEAZĂ. Costul resetării biţilor în cadrul ciclului **cât timp** este plătit cu dolarii aflaţi pe biţii care sunt resetaţi. Cel mult un bit este setat, pe linia 6 a procedurii INCREMENTEAZĂ, şi, prin urmare, costul amortizat al

unei operaţii INCREMENTEAZĂ este de cel mult 2 dolari. Deoarece numărul de 1 din contor nu este niciodată negativ, suma de credit este întotdeauna nenegativă. Rezultă, că pentru n operaţii INCREMENTEAZĂ, costul amortizat total este $O(n)$, care mărgineşte costul real total.

Exerciţii

18.2-1 Se efectuează n operaţii asupra unei stive a cărei mărime este întotdeauna cel mult egală cu k. După fiecare k operaţii, se copiază întreaga stivă în scopul salvării ei. Arătaţi că, pentru n operaţii de stivă, inclusiv salvarea ei, costul este $O(n)$ dacă stabilim în mod judicios costuri amortizate pentru diferitele operaţii de stivă.

18.2-2 Reluaţi exerciţiul 18.1-3 folosind metoda de cotare.

18.2-3 Să presupunem că dorim nu doar să incrementăm contorul, dar să îl şi resetăm la zero (adică să iniţializăm toţi biţii cu 0). Arătaţi cum poate fi implementat un contor sub forma unui vector de biţi, astfel încât orice secvenţă de n operaţii INCREMENTEAZĂ şi RESET să necesite timpul $O(n)$, presupunând că iniţial contorul este zero. (*Indicaţie:* Folosiţi un pointer la cifra cea mai semnificativă egală cu 1.)

18.3. Metoda de potenţial

În loc de a considera efortul plătit la început, sub forma unui credit repartizat anumitor obiecte din structura de date, ***metoda de potenţial*** a analizei amortizate tratează acest efort ca "energie potenţială" sau, pe scurt, "potenţial". Acesta va fi folosit pentru a plăti următoarele operaţii. Potenţialul este asociat structurii de date considerată ca un tot, în loc să fie repartizat anumitor obiecte din structura de date.

Descriem, în continuare, cum lucrează metoda de potenţial. Plecăm de la o structură de date iniţială D_0 asupra căreia se execută n operaţii. Pentru fiecare $i = 1, 2, \ldots, n$, fie c_i costul real al operaţiei i şi fie D_i structura de date obţinută din D_{i-1} după aplicarea operaţiei i. O ***funcţie de potenţial*** Φ ataşează fiecărei structuri de date D_i numărul real $\Phi(D_i)$, care este ***potenţialul*** asociat structurii de date D_i. ***Costul amortizat*** $\widehat{c}_i$ al operaţiei i pentru funcţia de potenţial Φ este definit prin

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}). \tag{18.1}$$

Prin urmare, costul amortizat al fiecărei operaţii este costul real plus creşterea de potenţial datorată operaţiei. Conform ecuaţiei (18.1), costul amortizat total al celor n operaţii este:

$$\sum_{i=1}^{n} \widehat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \tag{18.2}$$

A doua relaţie rezultă din ecuaţia (3.7), deoarece $\Phi(D_i)$ telescopează.

Dacă putem defini o funcţie de potenţial Φ, astfel încât $\Phi(D_n) \geq \Phi(D_0)$, atunci costul amortizat total $\sum_{i=1}^{n} \widehat{c}_i$ este o margine superioară pentru costul real total. În practică, nu ştim întotdeauna câte operaţii vor fi efectuate. De aceea, dacă cerem ca $\Phi(D_i) \geq \Phi(D_0)$ pentru orice i, atunci garantăm, ca în metoda de cotare, că plătim în avans. De multe ori, este convenabil să

definim $\Phi(D_0)$ ca fiind 0 şi să arătăm că $\Phi(D_i) \geq 0$ pentru toţi i. (Vezi exerciţiul 18.3-1 pentru a deprinde un mod simplu de tratare a cazurilor în care $\Phi(D_0) \neq 0$).

Intuitiv, dacă diferenţa de potenţial $\Phi(D_i) - \Phi(D_{i-1})$ a operaţiei i este pozitivă, atunci costul amortizat $\widehat{c_i}$ reprezintă o supracotare a operaţiei i, iar potenţialul structurii de date creşte. Dacă diferenţa de potenţial este negativă, atunci costul amortizat reprezintă o subcotare a operaţiei i, iar costul real al operaţiei este plătit prin descreşterea potenţialului.

Costurile amortizate definite de ecuaţiile (18.1) şi (18.2) depind de alegerea funcţiei de potenţial Φ. Funcţii de potenţial diferite pot conduce la costuri amortizate diferite, care rămân însă margini superioare ale costurilor reale. De multe ori are loc o "negociere" la alegerea funcţiei de potenţial: cea mai bună funcţie de potenţial depinde de limitele de timp dorite.

Operaţii de stivă

Pentru ilustrarea metodei de potenţial, ne întoarcem din nou la exemplul cu operatorii de stivă PUNE-ÎN-STIVĂ, SCOATE-DIN-STIVĂ şi SCOATERE-MULTIPLĂ-DIN-STIVĂ. Definim funcţia de potenţial Φ pe stivă ca fiind numărul de obiecte din stivă. Pentru stiva vidă, de la care se pleacă, avem $\Phi(D_0) = 0$. Deoarece numărul de elemente din stivă nu este niciodată negativ, stiva D_i la care se ajunge după operaţia i are un potenţial nenegativ şi, ca urmare,

$$\Phi(D_i) \geq 0 = \Phi(D_0).$$

Rezultă că pentru n operaţii costul amortizat total în raport cu Φ reprezintă o margine superioară a costului real.

Calculăm, în continuare, costurile amortizate ale diferitelor operaţii de stivă. Dacă operaţia i este efectuată pe o stivă conţinând s obiecte şi are tipul PUNE-ÎN-STIVĂ, atunci diferenţa de potenţial este

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1.$$

Conform ecuaţiei (18.1), costul amortizat al operaţiei PUNE-ÎN-STIVĂ este

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$

Să presupunem acum că operaţia i efectuată pe stivă este SCOATERE-MULTIPLĂ-DIN-STIVĂ(S, k) şi că un număr de $k' = \min(k, s)$ obiecte sunt extrase din stivă. Costul real al operaţiei este k', iar diferenţa de potenţial este

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Deci costul amortizat al operaţiei SCOATERE-MULTIPLĂ-DIN-STIVĂ este

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0.$$

În mod analog, obţinem un cost amortizat egal cu 0 pentru o operaţie SCOATE-DIN-STIVĂ.

Costurile amortizate ale fiecăreia dintre cele trei operaţii este $O(1)$ şi, deci, costul amortizat total al unei secvenţe de n operaţii este $O(n)$. Deoarece ştim că $\Phi(D_i) \geq \Phi(D_0)$, costul amortizat total a n operaţii constituie o margine superioară pentru costul real total. Costul celor n operaţii, în cazul cel mai defavorabil, este egal cu $O(n)$.

Incrementarea contorului binar

Pentru a prezenta încă un exemplu de aplicare a metodei de potenţial, vom considera din nou problema incrementării unui contor binar. De această dată vom defini potenţialul contorului după i operaţii INCREMENTEAZĂ ca fiind b_i = numărul de cifre 1 din contor după operaţia i.

Să calculăm acum costul amortizat al operaţiei INCREMENTEAZĂ. Să presupunem că cea de a i-a operaţie INCREMENTEAZĂ resetează t_i biţi. Costul real al operaţiei este cel mult $t_i + 1$ deoarece, pe lângă resetarea a t_i biţi, este setat la 1 cel mult un bit. Numărul de cifre 1 din contor, după a i-a operaţie INCREMENTEAZĂ, este egal cu $b_i \leq b_{i-1} - t_i + 1$, iar diferenţa de potenţial este

$$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i.$$

Drept urmare, costul amortizat este

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2.$$

Contorul porneşte de la zero, deci $\Phi(D_0) = 0$. Cum $\Phi(D_i) \geq 0$ pentru orice i, costul amortizat total al unei secvenţe de n operaţii INCREMENTEAZĂ este o margine superioară pentru costul real total; rezultă că, în cazul cel mai defavorabil, costul a n operaţii INCREMENTEAZĂ este $O(n)$.

Metoda de potenţial ne furnizează o modalitate simplă de analiză a contorului chiar şi în cazul în care acesta nu porneşte de la zero. Iniţial, există b_0 de 1, iar după n operaţii INCREMENTEAZĂ sunt b_n de 1, unde $0 \leq b_0$, $b_n \leq k$. Putem rescrie ecuaţia (18.2) astfel

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \widehat{c_i} - \Phi(D_n) + \Phi(D_0). \tag{18.3}$$

Avem $\widehat{c_i} \leq 2$ pentru orice $1 \leq i \leq n$. Cum $\Phi(D_0) = b_0$ şi $\Phi(D_n) = b_n$, costul real total a n operaţii INCREMENTEAZĂ este

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} 2 - b_n + b_0 = 2n - b_n + b_0.$$

În particular, ţinând cont de faptul că $b_0 \leq k$, dacă executăm cel puţin $n = \Omega(k)$ operaţii INCREMENTEAZĂ, costul real total va fi $O(n)$, indiferent de valoarea iniţială a contorului.

Exerciţii

18.3-1 Fie Φ o funcţie de potenţial cu $\Phi(D_i) \geq \Phi(D_0)$ pentru orice i, dar $\Phi(D_0) \neq 0$. Arătaţi că există o funcţie de potenţial Φ' astfel încât $\Phi'(D_0) = 0, \Phi'(D_i) \geq 0$ pentru orice $i \geq 1$, iar costurile amortizate corespunzătoare lui Φ' sunt aceleaşi ca şi costurile amortizate corespunzătoare lui Φ.

18.3-2 Reluaţi exerciţiul 18.1-3 folosind pentru analiză o metodă de potenţial.

18.3-3 Să considerăm o structură de date de tip ansamblu binar cu n elemente, pentru care instrucţiunile INSEREAZĂ şi EXTRAGE-MINIM cer timpul $O(\lg n)$ în cazul cel mai defavorabil. Determinaţi o funcţie de potenţial Φ astfel încât costul amortizat al procedurii INSEREAZĂ este $O(\lg n)$, costul amortizat al procedurii EXTRAGE-MINIM este $O(1)$ şi demonstraţi acest lucru.

18.3-4 Care este costul total al execuţiei a n operaţii PUNE-ÎN-STIVĂ, SCOATE-DIN-STIVĂ şi SCOATERE-MULTIPLĂ-DIN-STIVĂ pentru o stivă care conţine iniţial s_0 obiecte, iar în final s_n obiecte?

18.3-5 Să considerăm un contor care nu porneşte de la 0, ci pleacă de la un număr a cărui reprezentare în baza 2 conţine b de 1. Arătaţi că, pentru execuţia a n operaţii INCREMENTEAZĂ costul este $O(n)$ dacă $n = \Omega(b)$. (Valoarea b nu trebuie considerată ca fiind o constantă.)

18.3-6 Arătaţi cum poate fi implementată o coadă folosind două stive (exerciţiul 11.1-6), astfel încât costul amortizat al fiecărei operaţii PUNE-ÎN-COADĂ şi a fiecărei operaţii SCOATE-DIN-COADĂ să fie $O(1)$.

18.4. Tabele dinamice

În unele aplicaţii, nu ştim de la început câte obiecte vor fi memorate într-un tablou. Alocând un anumit spaţiu pentru tablou, riscăm să descoperim mai târziu că acest spaţiu este insuficient. Trebuie să facem o realocare pentru un nou tablou de dimensiuni mai mari şi să copiem toate elementele din tabloul iniţial în tabloul nou creat. În mod similar, dacă mai multe obiecte au fost şterse dintr-un tablou, poate fi convenabil să creăm un nou tablou, de dimensiuni mai mici, şi să copiem tabloul curent în cel nou creat. În această secţiune vom studia problema măririi (expandării) şi micşorării (comprimării) dinamice a unui tablou. Folosind analiza amortizată vom arăta că, pentru inserare şi ştergere, costul este $O(1)$, chiar şi atunci când costul real al unei operaţii este mai mare, atunci când are loc o mărire sau o micşorare a dimensiunilor. De asemenea vom arăta cum se poate garanta că spaţiul nefolosit dintr-un tablou dinamic nu va depăşi o subdiviziune constantă a spaţiului total.

Vom presupune că tabloul dinamic permite operaţiile INSEREAZĂ-TABLOU şi ŞTERGE-TABLOU. Operaţia INSEREAZĂ-TABLOU inserează în tablou un articol care ocupă o singură locaţie, adică spaţiul necesar memorării unui articol. În mod analog, operaţia ŞTERGE-TABLOU poate fi gândită ca fiind îndepărtarea unui articol din tablou, eliberând astfel o locaţie. Detaliile referitoare la structura de date folosită pentru organizarea tabloului nu au aici importanţă: putem folosi o stivă (secţiunea 11.1), un ansamblu (secţiunea 7.1) sau un tabel de dispersie (capitolul 12). Putem, de asemenea, să folosim un vector sau o colecţie de vectori pentru a implementa memorarea obiectelor, aşa cum s-a procedat în secţiunea 11.

Este convenabil să folosim un concept introdus în analiza dispersiei (capitolul 12). Definim ***factorul de încărcare*** $\alpha(T)$ al unui tablou T nevid ca fiind numărul de articole memorate în tablou, împărţit la dimensiunea (numărul de locaţii ale) tabloului. Tabloul vid (cu nici un articol) are dimensiunea 0 şi definim factorul său de încărcare ca fiind 1. Dacă factorul de încărcare al unui tablou dinamic este mărginit inferior de o constantă, spaţiul nefolosit din tablou nu depăşeşte niciodată o subdiviziune constantă a dimensiunii spaţiului total.

Începem prin a analiza un tablou dinamic asupra căruia se execută numai operaţii de inserare. Vom trece apoi la cazul general în care sunt permise atât inserări, cât şi ştergeri.

18.4.1. Expandarea tabloului

Presupunem că pentru alocarea memoriei pentru un tablou se foloseşte un vector de locaţii. Un tablou este considerat plin dacă toate locaţiile sale au fost folosite, adică atunci când factorul

său de încărcare este 1.[1] În unele medii software, dacă se face o încercare de a introduce un articol într-un tablou plin, singura alternativă posibilă este de a termina programul, împreună cu afişarea unui mesaj de eroare. Vom presupune aici că lucrăm într-un mediu modern care dispune de un sistem de gestiune a memoriei, care poate aloca şi elibera, la cerere, blocuri de memorie. Astfel, atunci când se încearcă inserarea unui articol într-un tablou plin, putem ***expanda*** (mări) tabloul prin alocarea unui nou tablou cu mai multe locaţii decât cel curent şi prin copierea elementelor din tabloul curent în cel nou.

O metodă euristică uzuală este de a aloca un nou tablou, care să aibă de două ori mai multe locaţii decât precedentul. Dacă se efectuează doar inserări, factorul de încărcare este, întotdeauna, cel puţin egal cu $1/2$ şi, astfel, spaţiul disponibil nu depăşeşte niciodată jumătate din spaţiul total alocat tabloului.

În pseudocodul care urmează, vom presupune că T este un obiect ce reprezintă tabloul. Câmpul $ref[T]$ conţine un pointer la blocul de memorie reprezentând tabloul. Câmpul $num[T]$ conţine numărul de articole din tablou, iar câmpul $dim[T]$ conţine numărul total de articole ce pot fi memorate în tablou. Iniţial, tabloul este vid: $num[T] = dim[T] = 0$.

INSEREAZĂ-TABLOU(T, x)
1: **dacă** $dim[T] = 0$ **atunci**
2: alocă pentru $ref[T]$ o locaţie de memorie
3: $dim[T] \leftarrow 1$
4: **dacă** $num[T] = dim[T]$ **atunci**
5: alocă pentru *tablou-nou* un număr de $2 \cdot dim[T]$ locaţii
6: inserează toate articolele din $ref[T]$ în $tablou - nou$
7: eliberează $ref[T]$
8: $ref[T] \leftarrow tablou_{nou}$
9: $dim[T] \leftarrow 2 \cdot dim[T]$
10: inserează x în $ref[T]$
11: $num[T] \leftarrow num[T] + 1$

Să observăm că avem, de fapt, două proceduri de "inserare": procedura INSEREAZĂ-TABLOU şi ***inserarea elementară*** în tablou, realizată în liniile 6 şi 10. Putem analiza timpul de execuţie al algoritmului INSEREAZĂ-TABLOU în funcţie de numărul de inserări elementare, atribuind costul 1 fiecărei inserări elementare. Vom presupune că timpul de execuţie real al lui INSEREAZĂ-TABLOU este liniar în funcţie de timpul cerut pentru inserarea unui singur articol, astfel încât efortul suplimentar pentru alocarea unui tablou iniţial în linia 2, este constant, iar efortul suplimentar pentru alocarea şi eliberarea de memorie, în liniile 5 şi 7, este mai mic decât costul transferării articolelor, realizat în linia 6. Vom numi ***expandare*** evenimentul care are loc atunci când este executată clauza **atunci** din liniile 5–9.

Să considerăm acum o secvenţă de n operaţii INSEREAZĂ-TABLOU efectuate asupra unui tablou iniţial vid. Care este costul c_i al operaţiei i? Dacă mai există loc în tabloul curent (sau dacă este vorba de prima operaţie), atunci $c_i = 1$, deoarece trebuie efectuată doar inserarea elementară din linia 10. Dacă tabloul curent este plin şi deci are loc o expandare, atunci $c_i = i$: inserarea elementară din linia 10 are costul 1, iar cele $i - 1$ mutări din tabloul curent în cel nou

[1] În unele situaţii, ca de exemplu pentru un tabel de dispersie cu adresare liberă, e posibil să dorim să considerăm un tablou ca fiind plin atunci când factorul său de încărcare este o constantă strict mai mică decât 1. (Vezi exerciţiul 18.4-2.)

creat, efectuate în linia 6, au fiecare costul 1. Dacă sunt executate n operaţii, costul pentru o operaţie, în cazul cel mai defavorabil, este $O(n)$, ceea ce conduce la o margine superioară $O(n^2)$ pentru timpul de execuţie total al celor n operaţii.

Această margine nu este cea mai convenabilă, deoarece costul expandării tabloului nu intervine foarte des în timpul efectuării celor n operaţii INSEREAZĂ-TABLOU. Mai precis, operaţia i conduce la o expandare numai dacă $i-1$ este o putere a lui 2. Costul amortizat al unei operaţii este, de fapt, $O(1)$, aşa cum se poate arăta dacă folosim metoda de agregare. Costul operaţiei i este:

$$c_i = \begin{cases} i & \text{dacă } i-1 \text{ este o putere a lui 2,} \\ 1 & \text{altfel.} \end{cases}$$

Costul total a n operaţii INSEREAZĂ-TABLOU este prin urmare:

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n,$$

deoarece există cel mult n operaţii de cost 1, iar costul celorlalte operaţii formează o progresie geometrică. Deoarece costul total a n operaţii INSEREAZĂ-TABLOU este $3n$, costul amortizat al unei singure operaţii este 3.

Folosind metoda de cotare, ne dăm seama mai uşor de ce costul amortizat al unei operaţii INSEREAZĂ-TABLOU este 3. Intuitiv, fiecare articol are de plătit pentru 3 inserări elementare: inserarea propriu-zisă în tablou, mutarea sa efectuată atunci când tabloul este expandat şi mutarea unui alt articol care a fost deja mutat o dată atunci când tabloul a fost expandat. De exemplu, să presupunem că, imediat după o expandare, dimensiunea tabloului este m. Atunci numărul efectiv de articole este $m/2$ şi tabloul nu conţine vreun credit. Cotăm cu 3 dolari fiecare inserare. Inserarea elementară care urmează imediat costă 1 dolar. Un al doilea dolar este plasat drept credit pe articolul inserat. Un al treilea dolar este plasat drept credit pe unul dintre cele $m/2$ articole existente deja în tablou. Umplerea tabloului cere $m/2$ inserări suplimentare şi, ca urmare, la momentul în care tabloul devine plin şi are deci m articole efective, fiecare articol are plasat pe el un dolar care va fi plătit la reinserarea sa în timpul următoarei expandări.

Pentru analiza unei secvenţe de n operaţii INSEREAZĂ-TABLOU, putem folosi şi metoda de potenţial; o vom folosi în secţiunea 18.4.2 pentru proiectarea unei operaţii ŞTERGE-TABLOU care va avea, de asemenea, costul amortizat $O(1)$. Începem prin a defini o funcţie de potenţial (a cărei valoare este 0 imediat după o expandare, dar creşte până la dimensiunea tabloului atunci când acesta devine plin), astfel încât putem plăti din potenţial pentru efectuarea expandării. Funcţia

$$\Phi(T) = 2 \cdot num[T] - dim[T] \tag{18.4}$$

reprezintă una dintre variantele posibile. Imediat după o expandare, avem $num[T] = dim[T]/2$ şi astfel $\Phi(T) = 0$, aşa cum dorim. În momentul dinaintea unei expandări, avem $num[T] = dim[T]$ şi, deci, $\Phi(T) = num[T]$, aşa cum dorim. Valoarea iniţială a potenţialului este 0 şi deoarece tabloul este întotdeauna plin cel puţin pe jumătate, $num[T] \ge dim[T]/2$, de unde rezultă că $\Phi(T)$ este întotdeauna nenegativă. Prin urmare suma costurilor amortizate a celor n operaţii INSEREAZĂ-TABLOU este o margine superioară pentru suma costurilor reale.

Pentru a analiza costul amortizat al celei de a i-a operaţii INSEREAZĂ-TABLOU, vom nota prin num_i numărul de articole memorate în tablou după această operaţie, prin dim_i dimensiunea

tabloului după această operaţie, iar prin Φ_i potenţialul după această operaţie. Iniţial avem $num_0 = 0$, $dim_0 = 0$, $\Phi_0 = 0$.

Dacă operaţia i nu conduce la o expandare, atunci $dim_i = dim_{i-1}$ şi costul amortizat al operaţiei este

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = 1 + (2 \cdot num_i - dim_i) - (2 \cdot num_{i-1} - dim_{i-1}) \\ &= 1 + (2 \cdot num_i - dim_i) - (2 \cdot (num_i - 1) - dim_i) = 3.\end{aligned}$$

Dacă operaţia i conduce la o expandare, atunci $dim_i/2 = dim_{i-1} = num_i - 1$, iar costul amortizat al operaţiei este

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = num_i + (2 \cdot num_i - dim_i) - (2 \cdot num_{i-1} - dim_{i-1}) \\ &= num_i + (2 \cdot num_i - (2 \cdot num_i - 2)) - (2 \cdot (num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) = 3.\end{aligned}$$

În figura 18.3 se arată cum variază valorile num_i, dim_i şi Φ_i în funcţie de i. Observaţi modul în care potenţialul este folosit pentru a plăti expandarea tabloului.

18.4.2. Expandarea şi contractarea tabloului

Pentru a implementa o operaţie ŞTERGE-TABLOU este suficient să îndepărtăm articolul specificat din tablou. Totuşi, uneori este de dorit să ***contractăm*** tabloul atunci când factorul de încărcare devine prea mic, pentru ca spaţiul nefolosit să nu fie exagerat de mare. Contractarea tabloului este analogă expandării sale: când numărul de articole din tablou scade prea mult, alocăm un nou tablou, mai mic, şi copiem articolele din vechiul tablou în noul tablou. Spaţiul de memorie ocupat de vechiul tablou poate fi acum eliberat prin transmiterea sa sistemului de gestiune a memoriei. Ideal, ar trebui satisfăcute întotdeauna două proprietăţi:

- factorul de încărcare al tabloului dinamic să fie mărginit inferior de o constantă şi
- costul amortizat al unei operaţii asupra tabloului să fie mărginit superior de o constantă.

Vom presupune că putem măsura costul în funcţie de inserările şi ştergerile elementare.

O strategie naturală pentru expandare şi contractare este cea de a dubla mărimea tabloului atunci când un articol trebuie introdus într-un tablou plin şi de a-i înjumătăţi dimensiunea atunci când o ştergere ar face ca tabloul să devină mai puţin plin decât jumătate. Această strategie garantează că factorul de încărcare al tabloului nu scade niciodată sub $1/2$, dar, din păcate, poate conduce la situaţia în care costul amortizat al unei operaţii să fie prea mare. Vom considera următorul scenariu. Efectuăm n operaţii asupra unui tablou T, unde n este o putere a lui 2. Primele $n/2$ operaţii sunt inserări care, conform unei analize anterioare, costă în total $\Phi(n)$. După această secvenţă de inserări, $num[T] = dim[T] = n/2$. Pentru ultimele $n/2$ operaţii, presupunem că ele se efectuează în următoarea ordine:

I, Ş, Ş, I, I, Ş, Ş, I, I, ...,

unde prin I am notat o inserare, iar prin Ş o ştergere. Prima inserare produce o expandare a tabloului la dimensiunea n. Următoarele două ştergeri conduc la o contractare a tabloului înapoi la dimensiunea $n/2$. Următoarele două inserări conduc la o expandare şi aşa mai departe. Există

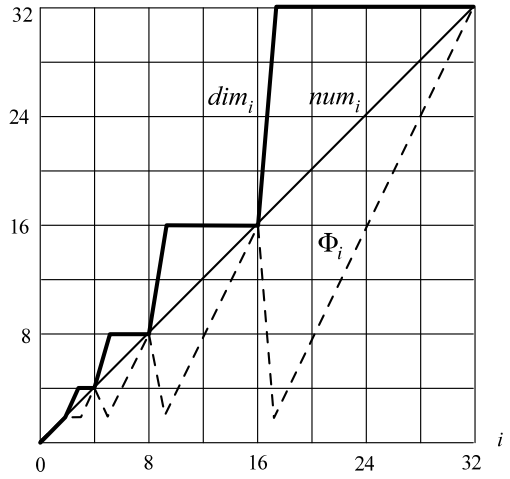


Figura 18.3 Efectul unei secvenţe de n operaţii INSEREAZĂ-TABLOU asupra numărului num_i de articole din tablou, numărului dim_i de locaţii din tablou şi potenţialului $\Phi_i = 2 \cdot num_i - dim_i$, fiecare fiind măsurat după operaţia i. Linia subţire îi corespunde lui num_i, linia groasă îi corespunde lui dim_i, iar cea punctată îi corespunde lui Φ_i. Observaţi că, în momentul dinaintea unei expandări, potenţialul a crescut până la numărul de articole din tablou, şi, de aceea, este suficient pentru a plăti pentru mutarea articolelor în noul tablou. Apoi potenţialul scade la 0, dar creşte imediat la 2 atunci când articolul care a produs expandarea este efectiv inserat.

$\Theta(n)$ expandări şi contractări, costul fiecăreia fiind $\Theta(n)$. Ca urmare, costul total al celor n operaţii este $\Theta(n^2)$, iar costul amortizat al unei operaţii este $\Theta(n)$.

Dificultatea legată de această strategie este evidentă: după o expandare nu se efectuează suficiente ştergeri pentru a putea plăti pentru contractare. În mod analog, după o contractare nu se efectuează suficiente inserări pentru a plăti pentru o expandare.

Putem îmbunătăţi această strategie permiţând factorului de încărcare să scadă sub 1/2. Mai precis, vom dubla în continuare dimensiunea tabloului atunci când un articol urmează a fi inserat într-un tablou plin, dar vom înjumătăţi dimensiunea tabloului atunci când o ştergere ar face ca el să fie plin sub un sfert, în loc de pe jumătate plin ca înainte. Factorul de încărcare al tabloului este, deci, mărginit inferior de constanta 1/4. După o expandare, factorul de încărcare devine 1/2. Astfel, jumătate din articolele tabloului trebuie şterse înainte ca să aibă loc o contractare, deoarece contractarea intervine numai când factorul de încărcare scade sub 1/4. Analog, după o contractare factorul de încărcare al tabloului este de asemenea 1/2. Din acest motiv, numărul de articole din tablou trebuie să se dubleze prin inserări pentru a urma o contractare, deoarece contractarea intervine doar dacă factorul de încărcare devine mai mare decât 1.

Nu vom prezenta codul pentru ŞTERGE-TABLOU, deoarece este analog cu INSEREAZĂ-TABLOU. Totuşi, în analiza pe care o facem este convenabil să presupunem că dacă numărul de articole din tablou devine 0, atunci memoria alocată tabloului este eliberată. Cu alte cuvinte, dacă $num[T] = 0$, atunci $dim[T] = 0$.

Putem folosi acum metoda de potenţial pentru a analiza costul unei secvenţe de n operaţii INSEREAZĂ-TABLOU şi ŞTERGE-TABLOU. Începem prin a defini o funcţie de potenţial Φ care ia valoarea 0 imediat după ce are loc o expandare şi se măreşte atunci când factorul de încărcare

creşte la 1 sau scade la 1/4. Vom defini factorul de încărcare al unui tablou T nevid prin $\alpha(T) = num[T]/dim[T]$. Deoarece pentru un tablou vid avem $num[T] = dim[T] = 0$ şi $\alpha(T) = 1$, vom avea întotdeauna $num[T] = \alpha(t) \cdot dim[T]$, indiferent dacă tabloul este vid sau nu. Vom folosi următoarea funcţie de potenţial:

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - dim[T] & \text{dacă } \alpha(T) \geq 1/2, \\ dim[T]/2 - num[T] & \text{dacă } \alpha(T) < 1/2. \end{cases} \tag{18.5}$$

Să observăm că funcţia de potenţial pentru un tablou vid este 0 şi că potenţialul nu este niciodată negativ. Ca urmare, costul amortizat total al unei secvenţe de operaţii relative la Φ este o margine superioară pentru costul real.

Înainte de a trece la o analiză mai detaliată, ne oprim pentru a remarca unele proprietăţi ale funcţiei de potenţial. Să observăm că, dacă factorul de încărcare este 1/2, atunci potenţialul este 0. Când el devine 1, avem $dim[T] = num[T]$, de unde $\Phi(T) = num[T]$ şi deci potenţialul este suficient pentru a plăti pentru expandare atunci când este inserat un articol. Când factorul de încărcare este 1/4, avem $dim[T] = 4 \cdot num[T]$, de unde rezultă că $\Phi(T) = num[T]$ şi, deci, potenţialul este suficient pentru a plăti pentru o contractare atunci când un articol este şters. Figura 18.4 ilustrează modul în care se comportă potenţialul pentru o secvenţă de operaţii.

Pentru a analiza o secvenţă de n operaţii INSEREAZĂ-TABLOU şi ŞTERGE-TABLOU, notăm prin c_i costul real al operaţiei i, prin $\widehat{c_i}$ costul său amortizat relativ la Φ, prin num_i numărul de articole memorate în tablou după operaţia i, prin dim_i mărimea tabloului după operaţia i, prin α_i factorul de încărcare al tabloului după operaţia i, iar prin Φ_i potenţialul după operaţia i. Iniţial, $num_0 = 0$, $dim_0 = 0$, $\alpha_0 = 1$, şi $\Phi_0 = 0$.

Începem cu cazul în care operaţia i este INSEREAZĂ-TABLOU. Dacă $\alpha_{i-1} \geq 1/2$, atunci analiza este identică cu cea pentru expandarea tabloului, prezentată în secţiunea 18.4.1. Indiferent dacă tabloul se expandează sau nu, costul amortizat $\widehat{c_i}$ al operaţiei este cel mult egal cu 3. Dacă $\alpha_{i-1} < 1/2$, tabloul nu poate fi expandat ca urmare a efectuării unei operaţii, deoarece expandarea se produce numai dacă $\alpha_{i-1} = 1$. Dacă şi $\alpha_i < 1/2$, costul amortizat al operaţiei i este

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = 1 + (dim_i/2 - num_i) - (dim_{i-1}/2 - num_{i-1}) \\ &= 1 + (dim_i/2 - num_i) - (dim_i/2 - (num_i - 1)) = 0. \end{aligned}$$

Dacă $\alpha_{i-1} < 1/2$ dar $\alpha_i \geq 1/2$, atunci

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = 1 + (2 \cdot num_i - dim_i) - (dim_{i-1}/2 - num_{i-1}) \\ &= 1 + (2 \cdot (2/num_{i-1} + 1) - dim_{i-1}) - (dim_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2} \cdot dim_{i-1} + 3 = 3 \cdot \alpha_{i-1} \cdot dim_{i-1} - \frac{3}{2} \cdot dim_{i-1} + 3 \\ &< \frac{3}{2} \cdot dim_{i-1} - \frac{3}{2} \cdot dim_{i-1} + 3 = 3. \end{aligned}$$

Rezultă că operaţia INSEREAZĂ-TABLOU are un cost amortizat mai mic sau egal cu 3.

Ne întoarcem acum la cazul în care operaţia i este ŞTERGE-TABLOU. În acest caz, $num_i = num_{i-1} - 1$. Dacă $\alpha_{i-1} < 1/2$, trebuie să luăm în considerare situaţia în care are loc o contractare. Dacă aceasta are loc, atunci $dim_i = dim_{i-1}$ şi costul amortizat al operaţiei este

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = 1 + (dim_i/2 - num_i) - (dim_{i-1}/2 - num_{i-1}) \\ &= 1 + (dim_i/2 - num_i) - (dim_i/2 - (num_i + 1)) = 2. \end{aligned}$$

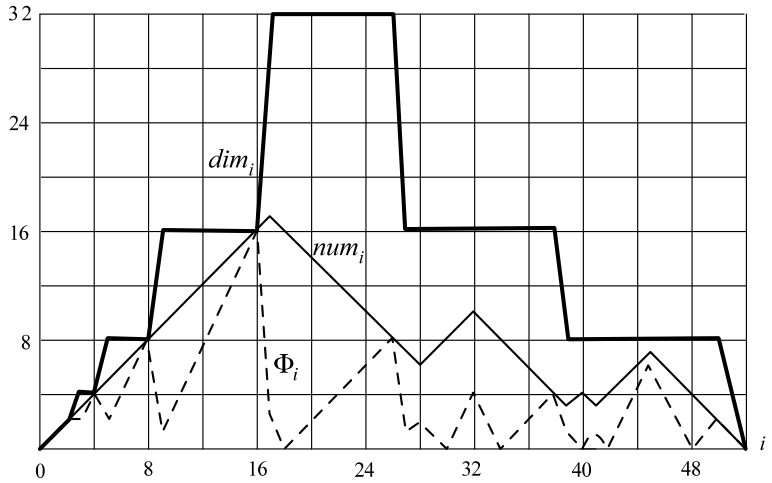


Figura 18.4 Efectul unei secvenţe de n operaţii INSEREAZĂ-TABLOU şi ŞTERGE-TABLOU asupra numărului num_i de articole din tablou, numărului dim_i de locaţii din tablou, potenţialului

$$\Phi_i = \begin{cases} 2 \cdot num_i - dim_i & \text{dacă } \alpha_i \geq 1/2, \\ dim_i/2 - num_i & \text{dacă } \alpha_i < 1/2, \end{cases}$$

fiecare fiind măsurat după operaţia i din secvenţă. Linia subţire corespunde lui num_i, linia groasă corespunde lui dim_i, iar linia punctată corespunde lui Φ_i. Să observăm că imediat înainte de expandare, potenţialul a crescut la numărul de articole din tablou şi de aceea el este suficient pentru a plăti pentru mutarea tuturor articolelor în noul tablou. Analog, imediat înainte de a avea loc o contractare, potenţialul a crescut la numărul de articole din tablou.

Dacă $\alpha_{i-1} < 1/2$ şi operaţia i nu implică o contractare, atunci costul real al operaţiei este $c_i = num_i + 1$, deoarece este şters un articol şi sunt mutate num_i articole. Avem $dim_i/2 = dim_{i-1}/4 = num_i + 1$, iar costul amortizat al operaţiei este

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi_i - \Phi_{i-1} = (num_i + 1) + (dim_i/2 - num_i) - (dim_{i-1}/2 - num_{i-1}) \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) = 1. \end{aligned}$$

Când operaţia i este ŞTERGE-TABLOU şi $\alpha_{i-1} \geq 1/2$, costul amortizat este, de asemenea, mărginit superior de o constantă. Analiza este propusă spre rezolvare cititorului în exerciţiul 18.4-3.

Recapitulând, deoarece costul amortizat al fiecărei operaţii este mărginit superior de o constantă, timpul real pentru orice secvenţă de n operaţii asupra unui tablou dinamic este $O(n)$.

Exerciţii

18.4-1 Argumentaţi de ce dacă $\alpha_{i-1} \leq 1/2$ şi $\alpha_i \leq 1/2$, atunci costul amortizat al unei operaţii INSEREAZĂ-TABLOU este 0.

18.4-2 Să presupunem că dorim să implementăm un tablou de dispersie dinamic, cu adresare liberă. De ce am putea considera tabloul plin dacă factorul său de încărcare atinge o valoare α strict mai mică decât 1? Descrieţi pe scurt cum trebuie făcută inserarea într-un asemenea tablou, astfel încât valoarea aşteptată a costului de amortizare pentru fiecare inserare să fie $O(1)$. De

ce valoarea aşteptată a costului de amortizare nu este în mod obligatoriu $O(1)$ pentru toate inserările?

18.4-3 Arătaţi că, dacă operaţia i pe un tablou dinamic este ŞTERGE-TABLOU şi $\alpha_{i-1} \geq 1/2$, atunci costul amortizat al operaţiei, în raport cu funcţia de potenţial (18.5), este mărginit superior de o constantă.

18.4-4 Să presupunem că, în loc să contractăm tabloul prin înjumătăţirea dimensiunii sale atunci când factorul de încărcare scade sub $1/4$, îl contractăm prin înmulţirea dimensiunii sale cu $2/3$ atunci când factorul de încărcare scade sub $1/3$. Folosind funcţia de potenţial

$$\Phi(T) = \mid 2 \cdot num[T] - dim[T] \mid,$$

arătaţi că, pentru o operaţie ŞTERGE-TABLOU, costul amortizat care foloseşte această strategie este mărginit superior de o constantă.

Probleme

18-1 Contor binar pe biţi inverşi

În capitolul 32 este prezentat un algoritm important, numit Transformarea Fourier Rapidă (pe scurt TFR). Primul pas al algoritmului TFR realizează o ***permutare cu inversarea biţilor*** a unui vector de intrare $A[0..n-1]$, a cărui lungime este $n = 2^k$ pentru un anumit k nenegativ. Această permutare interschimbă elementele de pe poziţiile ale căror reprezentări binare sunt una inversa celeilalte.

Putem exprima fiecare indice a ca o secvenţă de k biţi $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$ unde $a = \sum_{i=0}^{k-1} a_i 2^i$. Definim

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle) = \langle a_0, a_1 \ldots, a_{k-1} \rangle;$$

şi, deci,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i.$$

De exemplu, dacă $n = 16$ (adică $k = 4$), atunci $\text{rev}_k(3) = 12$, deoarece reprezentarea pe 4 biţi a lui 3 este 0011, a cărei inversă este 1100, care este tocmai reprezentarea pe 4 biţi a lui 12.

a. Presupunând că funcţia rev_k necesită timpul $\Theta(k)$, scrieţi un algoritm care să realizeze permutarea pe biţi inverşi a unui vector de lungime $n = 2^k$ în timpul $O(nk)$.

Putem folosi un algoritm bazat pe analiza amortizată pentru a îmbunătăţi timpul de execuţie a permutării cu inversarea biţilor. Folosim un "contor cu inversarea biţilor" şi o procedură INCREMENTEAZĂ-CU-INVERSAREA-BIŢILOR care, dată fiind o valoare a a contorului cu inversarea biţilor, produce $\text{rev}_k(\text{rev}_k(a) + 1)$. De exemplu, pentru $k = 4$ şi contorul cu inversarea biţilor iniţializat cu 0, apelurile succesive ale procedurii INCREMENTEAZĂ-CU-INVERSAREA-BIŢILOR produc secvenţa:

$$0000, 1000, 0100, 1100, 0010, 1010, ... = 0, 8, 4, 12, 2, 10, ...$$

b. Presupunem că pe calculatorul cu care lucrăm cuvintele sunt memorate ca valori pe k biţi şi că o unitate de timp este suficientă pentru a executa asupra lor operaţii de tipul deplasări la stânga şi la dreapta cu o valoare dată, ŞI şi SAU pe biţi etc. Scrieţi o implementare a lui INCREMENTEAZĂ-CU-INVERSAREA-BIŢILOR care să realizeze o permutare cu inversarea biţilor asupra unui vector cu n elemente în timpul $O(n)$.

c. Să presupunem că putem efectua o deplasare la stânga sau la dreapta cu un bit într-o unitate de timp. Mai este posibil să implementăm o permutare cu inversarea biţilor în timpul $O(n)$?

18-2 Varianta dinamică a căutării binare

Căutarea binară într-un vector ordonat necesită timp logaritmic, dar timpul necesitat de inserarea unui nou element este liniar în dimensiunea vectorului. Putem îmbunătăţi timpul pentru inserare prin memorarea mai multor vectori ordonaţi.

Mai precis, să presupunem că dorim să implementăm CĂUTARE şi INSEREAZĂ pe o mulţime cu n elemente. Fie $k = \lceil \lg(n+1) \rceil$ şi fie $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$ reprezentarea binară a lui n. Folosim k vectori ordonaţi $A_0, A_1, \ldots, A_{k-1}$, unde pentru fiecare $i = 0, 1, \ldots, k-1$ lungimea vectorului A_i este 2^i. Fiecare tablou poate fi doar vid sau plin, după cum $n_i = 1$ sau $n_i = 0$. Numărul total de elemente din cei k vectori va fi deci $\sum_{i=0}^{k-1} n_i 2^i = n$. Deşi fiecare vector în parte este ordonat, nu există o dependenţă anume între elementele din vectori diferiţi.

a. Arătaţi cum se poate realiza operaţia CĂUTARE pentru această structură de date. Analizaţi timpul de execuţie pentru cazul cel mai defavorabil.

b. Arătaţi cum poate fi inserat un element nou în această structură. Analizaţi timpii de execuţie pentru cazul cel mai defavorabil, precum şi timpul amortizat de execuţie.

c. Studiaţi modul în care poate fi implementată operaţia ŞTERGERE.

18-3 Arbori amortizaţi cu ponderi echilibrate

Să considerăm un arbore de căutare binar obişnuit, care are ataşată în plus, pentru fiecare nod x, valoarea $dim[x]$ care memorează numărul de chei din subarborele de rădăcină x. Fie α o constantă cu $1/2 \leq \alpha < 1$. Un nod x se numeşte ***α-echilibrat*** dacă

$$dim[stânga[x]] \leq \alpha \cdot dim[x] \text{ şi } dim[dreapta[x]] \leq \alpha \cdot dim[x]$$

Întregul arbore este ***α-echilibrat*** dacă fiecare nod din arbore este α-echilibrat. Următoarea abordare amortizată pentru a păstra caracteristica de arbori cu ponderi echilibrate a fost sugerată de G. Varghese.

a. Un arbore 1/2-echilibrat este, într-un anumit sens, atât de echilibrat pe cât este posibil. Fiind dat un nod x, într-un arbore binar de căutare oarecare, arătaţi cum poate fi reconstituit subarborele de rădăcină x astfel încât să devină 1/2-echilibrat. Algoritmul trebuie să se încadreze în timpul $\Theta(dim[x])$ şi poate folosi o memorie auxiliară de ordinul $O(dim[x])$.

b. Arătaţi că o căutare binară într-un arbore α-echilibrat având n noduri poate fi realizată în timpul $O(\lg n)$, pentru cazul cel mai defavorabil.

Pentru ceea ce urmează în această problemă, vom presupune că α este strict mai mare decât $1/2$. Presupunem că operaţiile INSEREAZĂ şi ŞTERGERE sunt implementate în modul uzual pentru un arbore de căutare binar având n noduri, cu următoarea excepţie: după fiecare astfel de operaţie, dacă toate nodurile din arbore nu mai sunt α-echilibrate, atunci subarborele având rădăcina în nodul de acest tip, situat pe cel mai de sus nivel în arbore, este "reconstituit" astfel încât devine $1/2$-echilibrat.

Vom analiza schema de reconstituire folosind metoda de potenţial. Pentru un nod x oarecare din arborele de căutare binară T, definim

$$\Delta(x) = \mid dim[stânga[x]] - dim[dreapta[x]] \mid$$

şi definim potenţialul lui T prin

$$\Phi(T) = c \cdot \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

unde c este o constantă suficient de mare, care depinde de α.

c. Arătaţi că orice arbore de căutare binar are potenţial nenegativ şi că un arbore $1/2$-echilibrat are potenţialul 0.

d. Să presupunem că m unităţi de potenţial sunt suficiente pentru a plăti reconstituirea unui subarbore având m noduri. Cât de mare trebuie să fie c, în funcţie de α, pentru ca timpul amortizat necesar reconstituirii unui subarbore care nu este α-echilibrat să fie $O(1)$.

e. Arătaţi că, pentru un arbore α-echilibrat având n noduri, inserarea şi ştergerea unui nod α-echilibrat necesită un timp amortizat de ordinul $O(\lg n)$.

Note bibliografice

Metoda de agregare a analizei amortizate a fost folosită de Aho, Hopcroft şi Ullman [4]. Tarjan [189] trece în revistă metodele de cotare şi de potenţial ale analizei amortizate şi prezintă mai multe aplicaţii. El atribuie paternitatea metodei de cotare mai multor autori, printre care M. R. Brown, R. E. Tarjan, S. Huddleston şi K. Mehlhorn. Tot el atribuie paternitatea metodei de potenţial lui D. D. Sleator. Termenul "amortizat" este datorat lui D. D. Sleator şi lui R. E. Tarjan.

V Structuri de date avansate

Introducere

În această parte reluăm, la un nivel mai avansat decât în partea a III-a, structurile de date care permit operaţii pe mulţimi dinamice. De exemplu, două dintre capitolele părţii folosesc, extensiv, tehnicile de analiză amortizată prezentate în capitolul 18.

Capitolul 19 prezintă structura de B-arbore, care este arborele echilibrat folosit la memorarea pe disc magnetic. Deoarece discul magnetic operează mult mai încet decât memoria RAM, performanţele B-arborilor vor fi măsurate nu numai prin timpul consumat cu operaţiile de căutare dinamice, ci şi prin numărul de accese la disc efectuate. Pentru fiecare operaţie, numărul de accese la disc este proporţional cu înălţimea B-arborelui.

Capitolele 20 şi 21 prezintă implementări ale heap-urilor fuzionabile, care permit operaţiile INSEREAZĂ, MINIMUM, EXTRAGE-MIN şi REUNEŞTE. Operaţia REUNEŞTE permite fuzionarea a două heap-uri. Pe structurile de date din aceste capitole se pot executa, de asemenea, operaţiile ŞTERGE şi DESCREŞTE-CHEIE.

Heap-urile binomiale, descrise în capitolul 20, execută fiecare dintre aceste operaţii, în cazul cel mai defavorabil, în timp $O(\lg n)$, unde n este numărul total de elemente din heap-ul de intrare (sau în cele două heap-uri de intrare în cazul REUNEŞTE). Se vede că operaţia REUNEŞTE din heap-ul binomial este superioară operaţiei definite pentru heap-ul binar prezentat în capitolul 7, acesta din urmă efectuând, în cel mai defavorabil caz, $\Theta(n)$ unităţi de timp pentru a fuziona.

Heap-urile Fibonacci din capitolul 21 sunt superioare heap-urilor binomiale, cel puţin în sens teoretic. Pentru măsurarea performanţelor heap-urilor Fibonacci se folosesc margini de timp amortizate. Se va vedea că operaţiile INSEREAZĂ, MINIMUM şi REUNEŞTE consumă în cazul heap-urilor Fibonacci, timp amortizat de numai $O(1)$, în timp ce EXTRAGE-MIN şi ŞTERGE consumă un timp amortizat de $O(\lg n)$. Cel mai mare avantaj al heap-urilor Fibonacci este acela că DESCREŞTE-CHEIE consumă un timp amortizat de numai $O(1)$. Operaţia DESCREŞTE-CHEIE definită pentru heap-urile Fibonacci este asimptotic mai slabă decât operaţiile similare la diverşii algoritmi specifici de grafuri, motiv pentru care aceştia o folosesc mai rar.

În sfârşit, capitolul 22 prezintă structuri de date pentru mulţimi disjuncte. Se dă un univers de n elemente care sunt grupate în mulţimi dinamice. Iniţial, fiecare element este singur într-o mulţime. Operaţia REUNEŞTE fuzionează două mulţimi, iar cererea GĂSEŞTE-MULŢIME identifică mulţimea care conţine un anumit element la un moment dat. Reprezentând fiecare mulţime printr-un arbore cu rădăcină, vom obţine operaţii surprinzător de rapide: o secvenţă de m operaţii se execută în timp $O(m\alpha(m, n))$, unde $\alpha(m, n)$ este o funcţie incredibil de încet crescătoare – în ipoteza că n reprezintă numărul de atomi din întregul univers, $\alpha(m, n)$ este cel mult 4. Analiza amortizată care determină această limită de timp este pe atât de complexă pe cât de simplă este structura de date. Capitolul 22 furnizează o limitare de timp interesantă şi relativ simplă.

Tematica abordată în această parte conţine nu numai exemple de structuri de date "avansate". Printre altele, mai sunt incluse şi următoarele:

- O structură de date, inventată de van Emde Boas [194], care admite operaţiile MINIMUM, MAXIMUM, INSEREAZĂ, ŞTERGE, CAUTĂ, EXTRAGE-MIN, EXTRACT-MAX, PREDECESOR, SUCCESOR. Acestea, în mulţimea de chei $\{1, 2, ..., n\}$ necesită, în cel mai defavorabil caz, timpul $(O(\lg \lg n))$.
- ***Arbori dinamici***, definiţi de către Sleator şi Tarjan [177] şi apoi reluaţi de Tarjan [188]. Cu ajutorul lor se întreţine o pădure de arbori cu rădăcini disjuncte. Fiecare arc din fiecare

arbore are ataşat, drept cost, un număr real. În arborii dinamici se pot efectua operaţiiîn vederea aflării părinţilor, rădăcinilor, costurilor arcelor, costului minim al unui drum de la un nod la o rădăcină. Arborii pot fi manevraţi prin tăierea de arce, modificarea costurilor arcelor de pe un drum de la un nod la o rădăcină, legarea unei rădăcini la un alt arbore etc. Una dintre implementările arborilor dinamici necesită un timp amortizat de $O(\lg n)$ pentru fiecare operaţie. O altă implementare, mai sofisticată, necesită timp $O(\lg n)$ în cel mai defavorabil caz.

- ***Arborii splay (Oblici)***, dezvoltaţi de către Sleator şi Tarjan [178] şi apoi reluaţi de Tarjan [188], sunt un tip de arbori binari de căutare care dau un timp amortizat $O(\lg n)$ pentru toate operaţiile standard. Una dintre aplicaţiile acestor arbori permite simplificarea arborilor dinamici.

- Structurile de date ***persistente*** permit atât cereri, cât şi, uneori, actualizări ale versiunilor vechi ale structurii. Sunt prezentate tehnicile de creare a structurilor de date înlănţuite persistente descrise de către Driscoll, Sarnak, Sleator şi Tarjan în [59]. Acestea sunt eficiente deoarece consumă foarte puţin timp şi spaţiu. Problema 14-1 dă un exemplu simplu de mulţime (de arbori) dinamică persistentă.

19 B-arbori

Un B-arbore este un arbore echilibrat, destinat căutării de informaţii memorate pe disc magnetic sau pe alt tip de suport adresabil direct. B-arborii sunt similari cu arborii roşu-negru (capitolul 14), dar ei minimizează operaţiile de intrare/ieşire disc.

Diferenţa esenţială faţă de arborii roşu-negru este aceea că fiecare nod dintr-un B-arbore poate avea un număr mare de fii, până la ordinul miilor. Astfel, "factorul de ramificare" poate fi foarte mare şi este, de regulă, determinat doar de caracteristicile unităţii de disc utilizate. Similaritatea cu arborii roşu-negru constă în faptul că ambii au înălţimea $O(\lg n)$, unde n este numărul de noduri, deşi înălţimea unui B-arbore poate fi considerabil mai mică din cauza factorului de ramificare mult mai mare. Din această cauză, B-arborii, pot fi, de asemenea, folosiţi pentru a implementa în timp $O(\lg n)$ diverse operaţii pe mulţimi dinamice.

B-arborii sunt o generalizare naturală a arborilor binari, după cum se poate vedea şi din figura 19.1 în care este ilustrat un B-arbore. Dacă un nod x al unui B-arbore conţine $n[x]$ chei, atunci x are $n[x]+1$ fii. Cheile nodului x sunt folosite pentru a separa domeniul de chei vizibile din x în $n[x]+1$ subdomenii, fiecare dintre acestea fiind vizibile prin unul dintre fiii lui x. Când se caută o cheie într-un B-arbore, există $n[x]+1$ posibilităţi de parcurgere, în funcţie de rezultatul comparării cu cele $n[x]$ chei memorate în nodul x.

În secţiunea 19.1 se dă definiţia exactă a B-arborelui şi se demonstrează că înălţimea lui creşte logaritmic cu numărul de noduri. Secţiunea 19.2 descrie căutarea şi inserarea într-un B-arbore, iar secţiunea 19.3 prezintă ştergerea. Înainte de acestea, vom discuta despre structurile de date, proiectate să lucreze pe disc magnetic şi vom evidenţia diferenţele de proiectare faţă de structurile de acces din memoria internă.

Structuri de date în memoria secundară

Există o mulţime de tehnologii dezvoltate pentru a permite exploatarea eficientă a memoriei unui sistem de calcul. ***Memoria internă (memoria principală)*** a unui sistem este compusă din chip-uri de memorie din silicon, fiecare dintre ele putând să păstreze circa un milion de biţi de informaţie. Costul acestei tehnologii este mai scump per bit de informaţie decât costul similar al tehnologiei magnetice: discuri sau benzi. Din această cauză sistemele de calcul au şi o ***memorie secundară***, rezidentă de obicei pe disc magnetic. Capacitatea acestui tip de suport depăşeşte

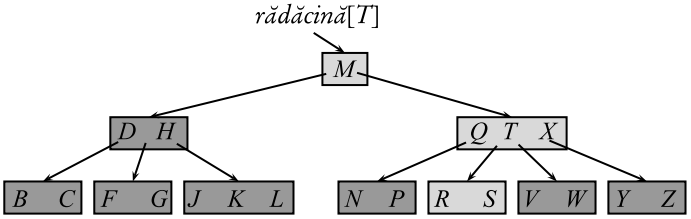


Figura 19.1 Un B-arbore în care cheile sunt consoane ale alfabetului latin. Un nod intern x conţine $n[x]$ chei şi are $n[x]+1$ fii. Toate frunzele sunt pe acelaşi nivel în arbore. Nodurile haşurate deschis indică traseul de căutare a literei R.

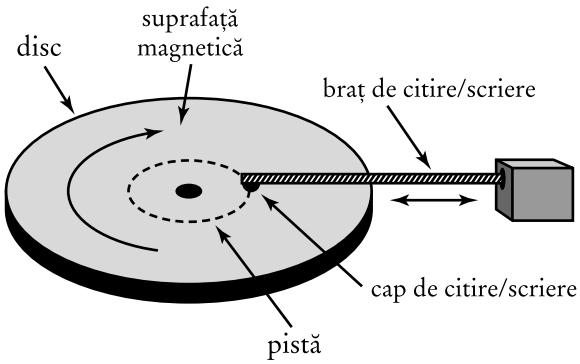


Figura 19.2 Schema unei unităţi de disc

cu câteva ordine de mărime capacitatea memoriei principale.

În figura 19.2 este prezentată, schematic, o unitate de disc magnetic clasică. Suprafaţa discului este acoperită cu o peliculă de material magnetizabil. Capul de citire/scriere poate citi/scrie de pe/pe această suprafaţă în timpul rotaţiei acesteia. Braţul de citire/scriere poate poziţiona capul la diferite distanţe de centrul discului. Atunci când braţul staţionează, partea din suprafaţă, care trece în timpul rotaţiei prin faţa unui cap de citire/scriere poartă numele de ***pistă***. Adesea, informaţiile înmagazinate pe o pistă sunt împărţite într-un număr fixat de bucăţi de lungime egală, numite ***pagini***; de obicei pagina unui disc are lungimea de 2048 octeţi. Pagina este unitatea de bază a schimbului dintre disc şi memoria internă. ***Timpul de acces*** la o pagină constă din timpul necesar poziţionării capului pe pistă, urmat de aşteptarea trecerii paginii prin faţa capului. Acest timp (de exemplu 20ms) este mult mai mare decât timpul necesar citirii/scrierii respective, deoarece poziţionarea este o activitate mecanică, pe când citirea/scrierea este una electronică. Din această cauză, o dată ce poziţionarea este corectă, citirea/scrierea unei mari cantităţi de informaţie de pe pista respectivă se face foarte rapid.

De regulă, timpul de citire a unei pagini este mai mare decât timpul necesar examinării şi prelucrării complete a informaţiilor citite. De aceea, în această secţiune vom analiza separat cele două componente ale timpului de execuţie:

- numărul de accese la disc;
- timpul de calcul al CPU.

Numărul de accese la disc va fi exprimat în funcţie de numărul de pagini citite sau scrise pe disc. Trebuie reţinut că accesul la disc nu este constant, ci depinde de distanţa dintre poziţia curentă şi pista ţintă, ca şi de starea curentă a mişcării de rotaţie. Pentru a simplifica lucrurile, vom aproxima numărul de pagini citite prin numărul de accese la disc.

Într-o aplicaţie clasică de B-arbori, cantitatea de informaţie manipulată depăşeşte cu mult capacitatea memoriei interne. Algoritmii specifici B-arborilor copiază, de pe disc în memorie, acele pagini de care este strictă nevoie şi le scriu înapoi pe cele care au fost modificate. Aceşti algoritmi presupun prezenţa simultană în memorie a unui număr constant de pagini, motiv pentru care dimensiunea memoriei principale nu limitează dimensiunile B-arborilor care sunt manevraţi.

În continuare, vom descrie, în pseudocod schema cadru de utilizare a discului. Fie x un pointer la un obiect. Dacă obiectul se află în memoria internă, atunci ne vom referi la el în mod obişnuit: $cheie[x]$. Dacă obiectul este momentan pe disc, atunci, mai întâi, trebuie să efectuăm operaţia CITEŞTE-DISC(x) pentru aducerea lui x în memoria internă înainte de a-i putea referi câmpurile. Dacă x este în memorie, atunci el nu mai trebuie citit. În mod analog, vom folosi operaţia SCRIE-DISC(x) dacă s-au efectuat modificări asupra componentelor lui x, pentru a le salva. Astfel, scenariul clasic de prelucrare a unui obiect este:

```
1: ...
2: x ← un pointer la un obiect
3: CITEŞTE-DISC(x)
4: operaţii de acces şi/sau de modificare a câmpurilor lui x
5: SCRIE-DISC(x)                    ▷ se omite dacă x nu s-a modificat
6: alte operaţii de acces care nu modifică câmpul lui x
7: ...
```

În fiecare moment, sistemul poate păstra în memoria internă numai un număr limitat de pagini. Algoritmii specifici B-arborilor presupun că există suficient spaţiu în memoria internă în acest scop.

Pe cele mai multe sisteme, timpii de execuţie ai algoritmilor de B-arbori sunt extrem de sensibili la numărul de operaţii CITEŞTE-DISC şi SCRIE-DISC. Din acest motiv ar fi normal ca un nod în B-arbore să aibă lungimea unei pagini disc. De asemenea, numărul de fii ai unui nod este bine să fie cât mai mare.

De obicei, factorul de ramificare este între 50 şi 2000, în funcţie de lungimea cheii şi de lungimea unei pagini. Un factor de ramificare mare reduce drastic înălţimea arborelui şi numărul de accese la disc pentru accesul la o cheie. De exemplu, în figura 19.3 este prezentat un B-arbore cu factorul de ramificare 1001 şi înălţimea 2, care permite memorarea unui miliard de chei. Cu toate acestea, deoarece nodul rădăcină poate fi păstrat permanent în memorie, sunt necesare numai ***două*** accese la disc pentru a găsi orice cheie din acest arbore.

19.1. Definiţia B-arborelui

Pentru a simplifica lucrurile, vom presupune, ca şi la arborii binari şi la cei roşu-negru, că "informaţia adiţională" asociată unei chei este memorată în acelaşi loc cu cheia. În practică, fiecărei chei îi este ataşat un pointer care reperează informaţia adiţională ataşată cheii. De asemenea, la B-arbori se obişnuieşte, tocmai pentru a creşte factorul de ramificare, ca, în nodurile interne să se memoreze numai chei şi pointeri spre fii, iar informaţia adiţională (pointer la ea) să fie înregistrată doar în frunze.

Un ***B-arbore*** T este un arbore cu rădăcină (cu rădăcina $rădăcină[T]$) care are următoarele proprietăţi:

1. Fiecare nod x are următoarele câmpuri:
   a. $n[x]$, numărul curent de chei memorate în x,
   b. cele $n[x]$ chei, memorate în ordine nedescrescătoare $cheie_1[x] \leq cheie_2[x] \leq \ldots \leq cheie_{n[x]}[x]$ şi

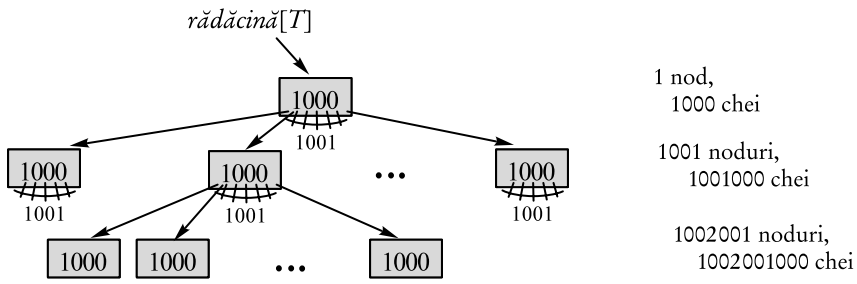


Figura 19.3 Un B-arbore de înălţime 2 conţine peste un miliard de chei. Fiecare nod intern sau frunză conţine 1000 de chei. Există 1001 noduri pe nivelul 1 şi peste un milion de frunze pe nivelul 2. Numărul scris în interiorul fiecărui nod x indică numărul $n[x]$ de chei din x.

   c. valoarea booleană $frunză[x]$, care este ADEVĂRAT dacă nodul x este frunză şi FALS dacă nodul este intern.

2. Dacă x este un nod intern, el mai conţine $n[x]+1$ pointeri spre fiii lui $c_1[x]$, $c_2[x]$, ..., $c_{n[x]+1}[x]$. Nodurile frunză nu au fii, astfel câmpurile lor c_i sunt nedefinite.

3. Cheile $cheie_i[x]$ separă domeniile de chei aflate în fiecare subarbore: dacă k_i este o cheie oarecare memorată într-un subarbore cu rădăcina $c_i[x]$, atunci

$$k_1 \leq cheie_1[x] \leq k_2 \leq cheie_2[x] \leq \ldots \leq cheie_{n[x]}[x] \leq k_{n[x]+1}.$$

4. Fiecare frunză se află la aceeaşi adâncime, care este înălţimea h a arborelui.

5. Există o limitare inferioară şi una superioară a numărului de chei ce pot fi conţinute într-un nod. Aceste margini pot fi exprimate printr-un întreg fixat $t \geq 2$, numit ***grad minim*** al B-arborelui:

   a. Fiecare nod, cu excepţia rădăcinii, trebuie să aibă cel puţin $t-1$ chei şi în consecinţă fiecare nod intern, cu excepţia rădăcinii, trebuie să aibă cel puţin t fii. Dacă arborele este nevid, atunci rădăcina trebuie să aibă cel puţin o cheie.

   b. Fiecare nod poate să conţină cel mult $2t-1$ chei. De aceea, orice nod intern poate avea cel mult $2t$ fii. Un nod cu $2t-1$ chei se va numi nod ***plin***.

Cel mai simplu B-arbore apare când $t = 2$. Orice nod intern poate avea 2, 3 sau 4 fii, motiv pentru care i se mai spune ***2-3-4 arbore***. Cu toate acestea, în practică, se folosesc valori mult mai mari pentru t.

Înălţimea unui B-arbore

Numărul de accese la disc cerute de cele mai multe operaţii în B-arbore este proporţional cu înălţimea B-arborelui. Vom analiza acum cazul cel mai defavorabil pentru înălţime.

Teorema 19.1 Dacă $n \geq 1$, atunci, pentru orice B-arbore T cu n chei de înălţime h şi gradul mimim $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2}.$$

Demonstraţie. Dacă un B-arbore are înălţimea h, va avea număr minim de chei dacă rădăcina conţine doar o cheie şi toate celelalte noduri câte $t-1$ chei. În acest caz, există două noduri pe nivelul 1, $2t$ noduri pe nivelul 2, $2t^2$ noduri pe nivelul 3 ş.a.m.d, $2t^{h-1}$ noduri pe nivelul h. Figura 19.4 prezintă un astfel de arbore pentru $h = 3$. De aceea, numărul n al cheilor satisface inegalitatea

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1,$$

ceea ce demonstrează teorema. ■

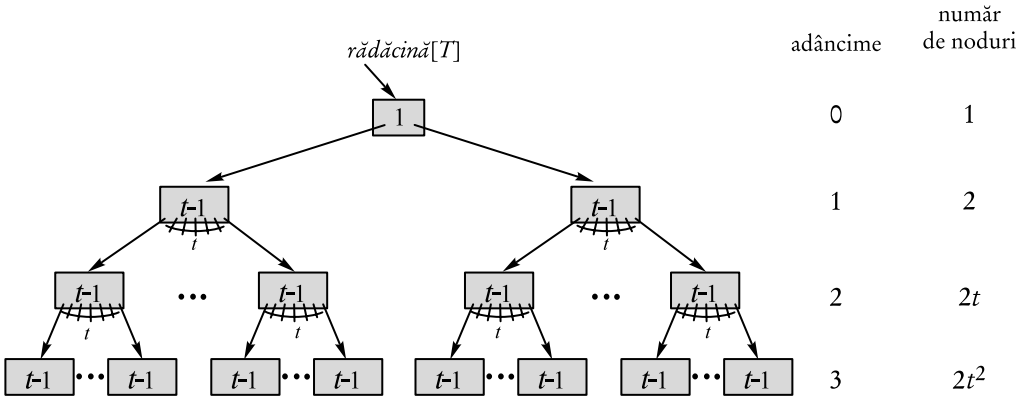


Figura 19.4 Un B-arbore de înălţime 3 care conţine numărul minim posibil de chei. În fiecare nod x este scris numărul efectiv de chei din nod, $n[x]$.

Aici se vede puterea B-arborilor în comparaţie cu arborii roşu-negru. Deşi înălţimea ambilor creşte proporţional cu $O(\lg n)$, t fiind o constantă, baza logaritmului la B-arbore poate fi, de multe ori, mai mare. De aceea, numărul nodurilor examinate la B-arbori scade cu factorul $\lg t$ faţă de arborii roşu-negru pentru majoritatea operaţiilor asupra arborilor. Astfel, numărul de accese la disc pentru B-arbori se reduce, automat, cu acest factor.

Exerciţii

19.1-1 De ce nu se permite gradul minim $t = 1$?

19.1-2 Pentru ce valori ale lui t, arborele din figura 19.1 este un B-arbore în conformitate cu definiţia?

19.1-3 Ilustraţi toţi B-arborii corecţi cu gradul minim 2 care reprezintă mulţimea $\{1, 2, 3, 4, 5\}$.

19.1-4 Determinaţi, în funcţie de gradul minim t, o margine superioară minimă a numărului de chei care pot fi memorate într-un B-arbore de înălţime h.

19.1-5 Descrieţi structura de date care rezultă când fiecare nod negru dintr-un arbore roşu-negru absoarbe fiul lui roşu, încorporând fiii lui cu el însuşi.

19.2. Operaţii de bază în B-arbore

În această secţiune, vom prezenta în detaliu, operaţiile CAUTĂ-B-ARBORE, CREEAZĂ-B-ARBORE şi INEREAZĂ-B-ARBORE. Pentru aceste proceduri facem următoarele convenţii:

- Rădăcina B-arborelui se află întotdeauna în memoria internă, astfel că, pentru ,ea nu este necesară operaţia CITEŞTE-DISC. Operaţia SCRIE-DISC se impune, totuşi, atunci când se modifică rădăcina.
- Fiecare nod transmis ca parametru trebuie să fi fost citit cu ajutorul procedurii CITEŞTE-DISC.

Procedurile prezentate sunt toate "dintr-o singură trecere", arborele fiind parcurs de la rădăcină spre frunze, fără reveniri.

Căutarea în B-arbore

Căutarea într-un B-arbore este analogă cu căutarea într-un arbore binar. Evident, în locul unei decizii "binare" care indică una dintre cele două căi posibile, aici vom avea de-a face cu o ramificare multiplă spre nodurile fii. Mai exact, la fiecare nod intern x, se ia o decizie din $n[x]+1$ alternative.

CAUTĂ-B-ARBORE este o generalizare naturală a procedurii CAUTĂ-ARBORE definită pentru arborii binari. CAUTĂ-B-ARBORE primeşte la intrare un pointer la nodul rădăcină x şi o cheie k ce trebuie căutată în subarborele de rădăcină x. La cel mai de sus nivel se va apela sub forma: CAUTĂ-B-ARBORE($rădăcină[T], k$). Dacă k este în B-arbore, atunci CAUTĂ-B-ARBORE returnează perechea ordonată (y, i) reprezentând un nod y şi un indice i, astfel încât $cheie_i[y] = k$. În caz contrar, returnează valoarea NIL.

Folosind o procedură de căutare liniară, liniile 1–3 găsesc cel mai mic i pentru care $k \leq cheie_i[x]$ dau lui i valoarea $n[x] + 1$ în caz contrar. Liniile 4–5 verifică dacă cheia a fost descoperită şi, în caz afirmativ, returnează valoarea corespunzătoare. Liniile 6–10 ori termină operaţia cu insucces (dacă x este o frunză), ori reapelează recursiv căutarea acestui fiu în subarborele corespunzător, după citirea prealabilă a acestuia.

În figura 19.1 se ilustrează operaţia de căutare CAUTĂ-B-ARBORE a cheii R într-un B-arbore. Sunt evidenţiate nodurile vizitate în acest caz.

La fel ca la procedura CAUTĂ-ARBORE definită pentru arbori binari, în timpul apelurilor recursive, sunt parcurse nodurile pe o cale de la rădăcină în jos. Din această cauză, numărul paginilor disc accesate de CAUTĂ-B-ARBORE sunt $\Theta(h) = \Theta(\log_t n)$, unde h este înălţimea B-arborelui, şi n numărul de chei. Deoarece $n[x] < 2t$, timpul consumat în ciclul **cât timp** din liniile 2–3, pentru fiecare nod, este $O(t)$ şi timpul total CPU este $O(th) = O(t \log_t n)$.

Crearea unui B-arbore vid

Pentru a construi un B-arbore T, mai întâi, se aplică procedura CREEAZĂ-B-ARBORE pentru a crea un nod rădăcină vid. Apoi, se apelează procedura INEREAZĂ-B-ARBORE pentru a adăuga câte o cheie nouă. Fiecare dintre aceste două proceduri foloseşte o procedură auxiliară ALOCĂ-NOD, care, în timp $O(1)$, alocă o pagină disc spre a fi folosită drept nod nou. Presupunem că nodul creat prin ALOCĂ-NOD nu necesită CITEŞTE-DISC deoarece pentru crearea unui nod nou nu sunt necesare informaţii de pe disc. Operaţia CREEAZĂ-B-ARBORE necesită $O(1)$ accesări de disc şi $O(1)$ timp CPU.

```
CAUTĂ-B-ARBORE(x, k)
 1: i ← 1
 2: cât timp i ≤ n[x] şi k > cheie_i[x] execută
 3:    i ← i + 1
 4: dacă i ≤ n[x] şi k = cheie_i[x] atunci
 5:    returnează (x, i)
 6: dacă frunză[x] atunci
 7:    returnează NIL
 8: altfel
 9:    CITEŞTE-DISC(c_i[x])
10:    returnează CAUTĂ-B-ARBORE(c_i[x], k)
```

```
CREEAZĂ-B-ARBORE(T)
1: x ← ALOCĂ-NOD()
2: frunză[x] ← ADEVĂRAT
3: n[x] ← 0
4: SCRIE-DISC(x)
5: rădăcină[T] ← x
```

Divizarea unui nod

Inserarea într-un B-arbore este, în mod esenţial, mai complicată decât inserarea unei chei într-un arbore binar de căutare. Una dintre operaţiile fundamentale folosite în timpul inserării este ***divizarea*** unui nod plin y (care are $2t-1$ chei) în două noduri în jurul ***cheii mediane*** $cheie_t[y]$, având câte $t-1$ chei fiecare. Cheia mediană se deplasează la locul potrivit în nodul părinte (care înainte de deplasare nu trebuie să fie plin) spre a repera fiecare dintre cei doi subarbori obţinuţi prin divizarea lui y. Dacă nodul y nu are părinte, atunci înălţimea B-arborelui creşte cu unu. Deci divizarea este cea care provoacă creşterea înălţimii.

Procedura DIVIDE-FIU-B-ARBORE primeşte ca intrare un nod intern x care *nu este plin* (presupusă a fi în memoria principală), un indice i şi un nod y astfel încât $y = c_i[x]$ este un fiu *plin* al lui x. Procedura divide nodul y în două şi rearanjează x astfel încât acesta va avea încă un fiu.

Figura 19.5 ilustrează acest proces. Nodul plin y este divizat după cheia lui mediană S care este mutată în nodul părinte x. Cheile mai mari decât S din y sunt mutate într-un nod nou z care devine un nou fiu al lui x.

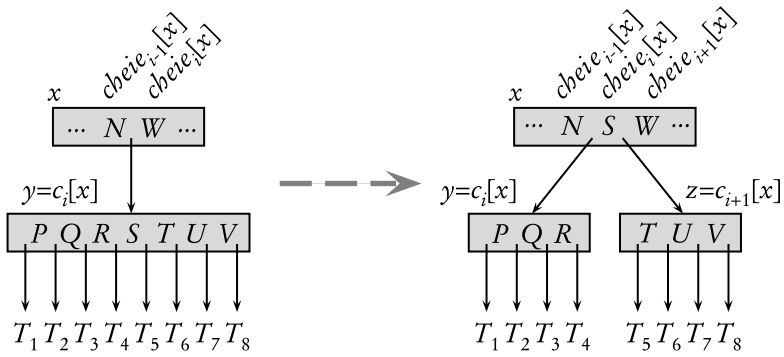


Figura 19.5 Divizarea unui nod cu $t = 4$. Nodul y este divizat în două noduri y şi z, iar cheia mediană S este mutată în părintele lui y.

DIVIDE-FIU-B-ARBORE(x, i, y)
1: $z \leftarrow$ ALOCĂ-NOD()
2: $frunză[z] \leftarrow frunză[y]$
3: $n[z] \leftarrow t - 1$
4: **pentru** $j \leftarrow 1, t-1$ **execută**
5: $\quad cheie_j[z] \leftarrow cheie_{j+t}[y]$
6: **dacă** nu $frunză[y]$ **atunci**
7: $\quad$ **pentru** $j \leftarrow 1, t$ **execută**
8: $\quad\quad c_j[z] \leftarrow c_{j+t}[y]$
9: $n[y] \leftarrow t - 1$
10: **pentru** $j \leftarrow n[x] + 1, i + 1, -1$ **execută**
11: $\quad c_{j+1}[x] \leftarrow c_j[x]$
12: $c_{i+1}[x] \leftarrow z$
13: **pentru** $j \leftarrow n[x], i, -1$ **execută**
14: $\quad cheie_{j+1}[x] \leftarrow cheie_j[x]$
15: $cheie_i[x] \leftarrow cheie_t[y]$
16: $n[x] \leftarrow n[x] + 1$
17: SCRIE-DISC(y)
18: SCRIE-DISC(z)
19: SCRIE-DISC(x)

Procedura DIVIDE-FIU-B-ARBORE lucrează într-o manieră obişnuită "cut and paste". Astfel y, al i-lea fiu al lui x, este nodul care va fi divizat. Nodul y are iniţial $2t-1$ fii, dar după divizare îi mai rămân doar primii $t-1$ fii. Nodul z "adoptă" ultimii $t-1$ fii ai lui y, în timp ce cheia mediană este deplasată în părintele x, spre a separa nodurile y şi z.

Liniile 1–8 creează nodul z, deplasează ultimele $t-1$ chei şi cei t subarbori corespunzători din nodul y. Linia 9 actualizează numărul de chei din y. Liniile 10–16 inserează z ca fiu al lui x, mută cheia mediană din y în x spre a separa nodurile y şi z şi actualizează numărul de chei din x. În sfârşit, liniile 17–19 cer scrierea paginilor disc modificate. Timpul CPU consumat de DIVIDE-FIU-B-ARBORE este $\Theta(t)$ datorită ciclurilor din liniile 4–5 şi 7–8 (celelalte cicluri au cel mult t iteraţii).

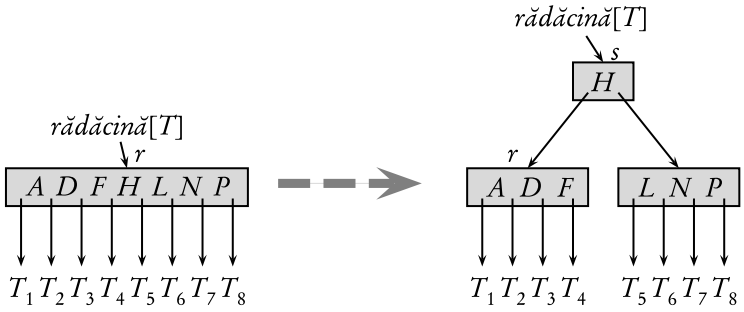


Figura 19.6 Divizarea rădăcinii cu $t = 4$. Nodul rădăcină r este divizat în două şi se creează un nou nod rădăcină s. Noul nod conţine cheia mediană din r şi ca fii cele două jumătăţi ale lui r. Înălţimea B-arborelui creşte cu 1 când rădăcina este divizată.

Inserarea unei chei într-un B-arbore

Operaţia de inserare a unei chei k într-un B-arbore de înălţime h se execută într-o singură parcurgere, coborând în arbore, şi cere $O(h)$ accese disc. Timpul CPU cerut este $O(th) = O(t \log_t n)$. Procedura INSEREAZĂ-B-ARBORE foloseşte procedura DIVIDE-FIU-B-ARBORE spre a se asigura că nici un descendent nu va deveni nod plin.

```
INSEREAZĂ-B-ARBORE(T, k)
 1: r ← rădăcină[T]
 2: dacă n[r] = 2t − 1 atunci
 3:    s ← ALOCĂ-NOD()
 4:    rădăcină[T] ← s
 5:    frunză[s] ←FALS
 6:    n[s] ← 0
 7:    c1[s] ← r
 8:    DIVIDE-FIU-B-ARBORE(s, 1, r)
 9:    INSEREAZĂ-B-ARBORE-NEPLIN(s, k)
10: altfel
11:    INSEREAZĂ-B-ARBORE-NEPLIN(r, k)
```

Liniile 3–9 tratează cazul în care nodul rădăcină r este plin: se creează o rădăcină nouă s cu doi descendenţi şi este divizată vechea rădăcină r. Divizarea rădăcinii este singura operaţie care determină creşterea înălţimii B-arborelui. În figura 19.6 este ilustrată această situaţie. Spre deosebire de arborii binari, care cresc în jos (spre frunze), B-arborele creşte în partea de sus, prelungind rădăcina. Procedura se încheie apelând INSEREAZĂ-B-ARBORE-NEPLIN pentru a insera cheia k într-un arbore cu o rădăcină care nu este plină. Această din urmă procedură parcurge arborele recursiv spre frunze, garantându-se faptul că, după parcurgere, nu vor rămâne noduri pline (dacă este cazul se va apela DIVIDE-FIU-B-ARBORE.

Procedura recursivă auxiliară INSEREAZĂ-B-ARBORE-NEPLIN inserează cheia k în nodul x despre care se presupune că nu este plin la momentul apelului. Împreună cu INSEREAZĂ-B-ARBORE, se asigură această condiţie pentru toate nodurile parcurse.

```
INSEREAZĂ-B-ARBORE-NEPLIN(x, k)
 1: i ← n[x]
 2: dacă frunză[x] atunci
 3:    cât timp i ≥ 1 şi k < cheie_i[x] execută
 4:       cheie_{i+1}[x] ← cheie_i[x]
 5:       i ← i − 1
 6:    cheie_{i+1}[x] ← k
 7:    n[x] ← n[x] + 1
 8:    SCRIE-DISC(x)
 9: altfel
10:    cât timp i ≥ 1 şi k < cheie_i[x] execută
11:       i ← i − 1
12:    i ← i + 1
13:    CITEŞTE-DISC(c_i[x])
14:    dacă n[c_i[x]] = 2t − 1 atunci
15:       DIVIDE-FIU-B-ARBORE(x, i, c_i[x])
16:       dacă k > cheie_i[x] atunci
17:          i ← i + 1
18:    INSEREAZĂ-B-ARBORE-NEPLIN(c_i[x], k)
```

Procedura INSEREAZĂ-B-ARBORE-NEPLIN lucrează după cum urmează. Liniile 3–8 inserează k în x când x este un nod frunză. Dacă x nu este frunză, atunci k trebuie inserată în frunza potrivită, într-un subarbore al lui x. În acest caz, liniile 9–12, determină fiul lui x de la care trebuie mers descendent. Linia 14 detectează dacă apare un nod plin, caz în care linia 15 apelează divizarea, iar în liniile 16–17 se alege nodul pe care se coboară în arbore, dintre cei doi fii obţinuţi prin divizare. Trebuie remarcat că, după incrementarea lui i din linia 17, nu mai trebuie apelat CITEŞTE-DISC($c_i[x]$) deoarece apelul recursiv asigură că fiul este deja citit în urma creării lui prin DIVIDE-FIU-B-ARBORE. Ca urmare a acţiunii liniilor 14–17, se garantează că nu vor mai apărea noduri pline. Linia 18 intră în recursivitate pentru a insera k în subarborele potrivit. În figura 19.7 se ilustrează câteva dintre cazurile care pot să apară la inserarea în B-arbore.

Numărul de accesări de disc efectuate de INSEREAZĂ-B-ARBORE este $O(h)$, unde h este înălţimea B-arborelui, deoarece între apelurile INSEREAZĂ-B-ARBORE-NEPLIN sunt efectuate numai $O(1)$ accese la disc. Timpul total CPU este $O(th) = O(t \log_t n)$. Deoarece INSEREAZĂ-B-ARBORE-NEPLIN foloseşte recursivitatea de coadă,ea poate fi transformată într-un ciclu **cât timp**, ceea ce arată că, în fiecare moment, numărul de pagini din memoria internă este $O(1)$.

Exerciţii

19.2-1 Arătaţi rezultatul inserării cheilor F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E, în această ordine, într-un B-arbore, iniţial vid. Desenaţi configuraţiile din arbore care preced divizări de noduri şi desenaţi configuraţia finală.

19.2-2 Explicaţi în ce condiţii pot fi (sunt) executate operaţii redundante de acces la disc la apelarea procedurii INSEREAZĂ-B-ARBORE. (O operaţie CITEŞTE-DISC este redundantă dacă pagina este deja în memorie, iar o operaţie SCRIE-DISC este redundantă dacă informaţia scrisă din memorie este identică cu cea de pe disc.)

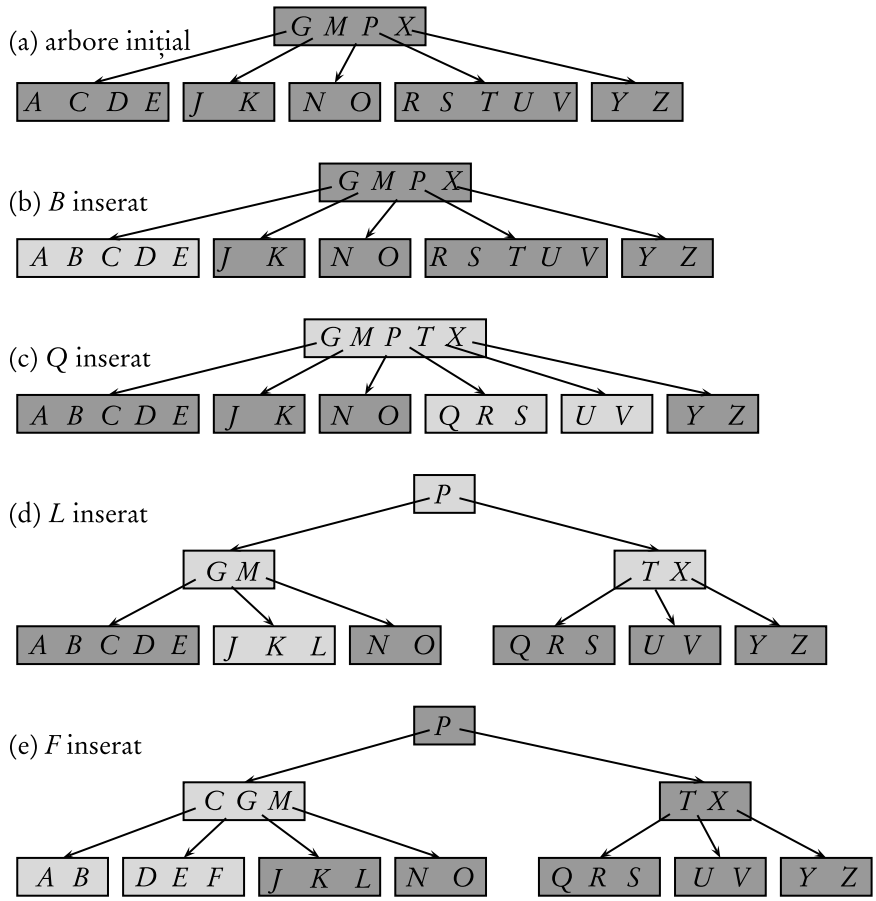


Figura 19.7 Inserarea de chei într-un B-arbore. Acesta are gradul minim $t = 3$, deci un nod poate avea cel mult 5 chei. Nodurile modificate prin inserări sunt haşurate mai deschis. **(a)** Arborele iniţial. **(b)** Rezultatul inserării cheii B – introducerea ei în frunză. **(c)** Introducerea cheii Q în arborele precedent: nodul $RSTUV$ este divizat în două noduri conţinând RS şi UV, cheia T este mutată în rădăcină, iar Q este inserat în partea stângă a nodului RS. **(d)** Rezultatul inserării cheii L în arborele precedent. Rădăcina este divizată, iar înălţimea B-arborelui creşte cu 1. Apoi L este inserat în frunza care conţine JK. **(e)** Rezultatul inserării cheii F în arborele anterior. Nodul $ABCDE$ este divizat înainte ca F să fie inserat în nodul DE.

19.2-3 Explicaţi cum se poate găsi cheia minimă memorată într-un B-arbore şi cum se poate găsi predecesorul unei chei memorate într-un B-arbore.

19.2-4 $\star$ Presupunem că cheile $\{1, 2, \ldots, n\}$ sunt inserate într-un B-arbore iniţial vid cu gradul minim 2. Câte noduri va avea în final acest B-arbore?

19.2-5 Deoarece nodurile frunză nu conţin pointeri la subarbori fii, se poate concepe folosirea în frunze a unei valori t diferite (mai mari) decât la nodurile interne pentru a ocupa mai eficient

paginile disc. Arătaţi cum se modifică procedurile de creare şi inserare într-un B-arbore în acest caz.

19.2-6 Presupunem că algoritmul CAUTĂ-B-ARBORE este implementat folosind, în interiorul nodurilor, căutarea binară în locul celei liniare. Arătaţi că, prin aceasta, timpul CPU este $O(\lg n)$, independent de t ca funcţie de n.

19.2-7 Presupunem că echipamentul disc permite să se aleagă arbitrar lungimea paginii, dar că timpul de citire a unei pagini, cu o lungime selectată, este $a + bt$, unde a şi b sunt constante date, iar t este gradul minim al B-arborelui. Descrieţi cum trebuie ales t, astfel încât să minimizeze (aproximativ) timpul de căutare în B-arbore. Căutaţi, de exemplu, o valoare optimă pentru t în cazul $a = 30$ milisecunde şi $b = 40$ microsecunde.

19.3. Ştergerea unei chei dintr-un B-arbore

Operaţia de ştergere dintr-un B-arbore este analogă celei de inserare, însă este puţin mai complicată. În loc să descriem complet algoritmii necesari, vom schiţa doar modul de lucru.

Presupunem că procedurii ŞTERGE-B-ARBORE i se cere să şteargă cheia k din subarborele de rădăcină x. Procedura trebuie să asigure (recursiv) faptul că numărul de chei din x este cel puţin t. Această cerinţă apare din faptul că, în urma unei ştergeri, se diminuează cu 1 numărul cheilor dintr-un nod, deci este posibil să nu mai poată avea t chei în el. În acest caz, poate avea loc un proces de ***fuziune*** (invers divizării), caz în care o cheie coboară într-un fiu înainte de a i se aplica acestuia (recursiv) procedura de ştergere. Dacă într-un astfel de proces rădăcina rămâne fără nici o cheie, deci cu un singur fiu, atunci se şterge rădăcina veche, iar unicul fiu devine noua rădăcină a arborelui, scăzând înălţimea arborelui cu 1 şi păstrând proprietatea arborelui cel puţin o cheie (doar dacă arborele nu este vid).

Figura 19.8 ilustrează câteva cazuri de ştergere de chei dintr-un B-arbore.

1. Dacă cheia k este într-un nod x care este frunză, atunci se şterge cheia k din nodul x.

2. Dacă cheia k este în nodul interior x, atunci:

   a. Dacă fiul y care precede cheia k are cel puţin t chei, atunci se caută predecesorul k' al cheii k în subarborele de rădăcină y. Se şterge k' şi se înlocuieşte k din x cu k', după care se aplică mai departe, recursiv, aceeaşi regulă. (Găsirea cheii k' şi ştergerea ei se poate face într-un singur pas.)

   b. În manieră simetrică, dacă fiul z care succede cheia k în nodul x are cel puţin t chei, atunci se caută succesorul k' al cheii k în subarborele de rădăcină z. Se şterge k' şi se înlocuieşte k din x cu k', după care se aplică mai departe, recursiv, aceeaşi regulă. (Găsirea cheii k' şi ştergerea ei se poate face într-un singur pas.)

   c. În caz contrar, dacă atât y cât şi z au numai câte $t-1$ chei, cele două noduri fuzionează în sensul că în y intră k şi toate cheile din z, după care y va conţine $2t-1$ chei. Apoi se eliberează nodul z şi recursiv, se şterge cheia k din y.

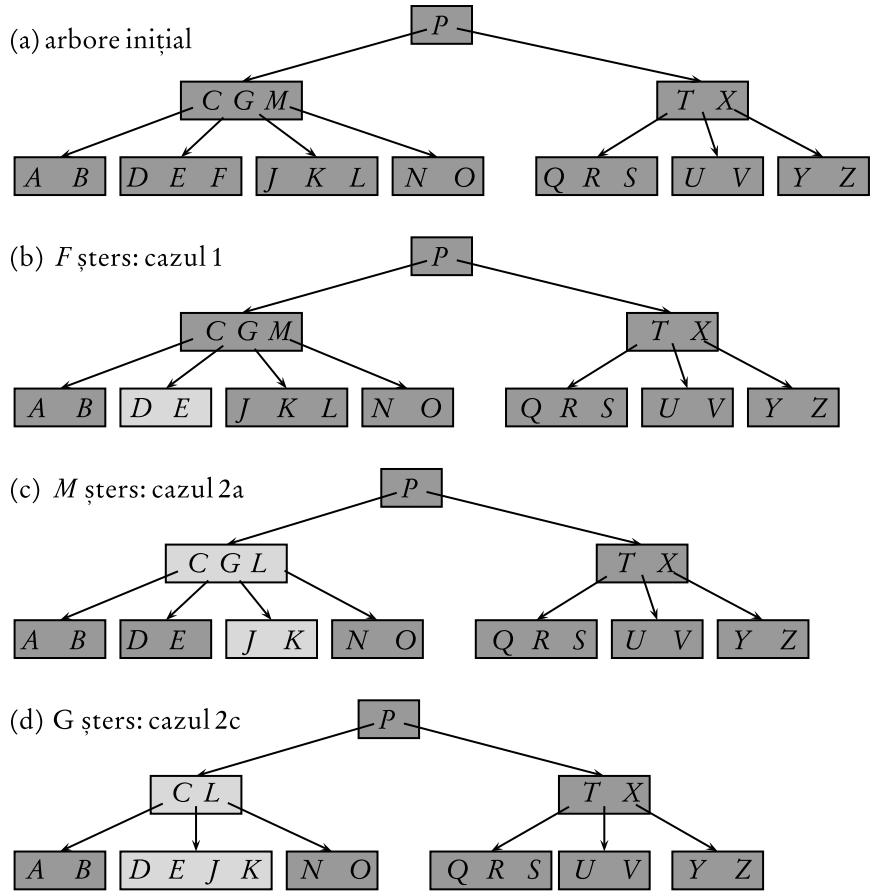


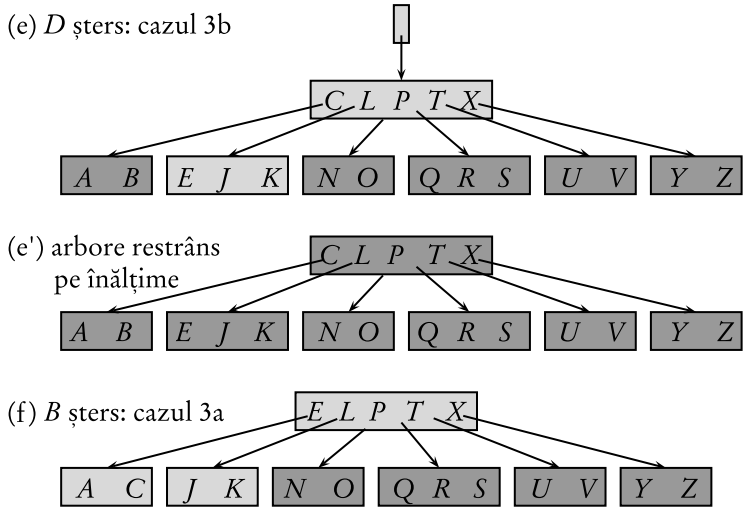
Figura 19.8 Ştergerea de chei dintr-un B-arbore. Gradul lui minim este $t = 3$ astfel că nodurile, cu excepţia rădăcinii, nu pot avea mai puţin de 2 chei. Nodurile modificate sunt haşurate mai deschis. **(a)** B-arborele din figura 19.7(e). **(b)** Ştergerea cheii F corespunde cazului 1: ştergere din frunză. **(c)** Ştergerea cheii M: cazul 2a, cheia L care îl precede pe M este mutată în vechiul loc al lui M. **(d)** Ştergerea cheii G: cazul 2c, cheia G este pusă mai jos şi se formează prin fuziune nodul $DEGJK$, după care G este şters din frunză (cazul 1). **(e)** Ştergerea cheii D: cazul 3b, recurenţa nu poate să coboare din nodul CL deoarece are numai două chei, deci P este împins în jos şi prin fuzionarea cu CL şi TX se obţine nodul $CLPTX$; apoi se şterge D din frunză (cazul 1). **(e')** După (d) se şterge rădăcina şi înălţimea scade cu 1. **(f)** Ştergerea cheii B: cazul 3a, C este mutat în vechiul loc al lui B şi E este mutat în locul lui C.

3. Dacă cheia k nu este prezentă în nodul intern x, se determină rădăcina $c_i[x]$ care indică subarborele în care se află cheia k, dacă aceasta se află în arbore. Dacă $c_i[x]$ are numai $t-1$ chei, se execută paşii 3a sau 3b, necesari pentru a se păstra în noduri numărul minim de t chei. Apoi se reaplică recursiv procedura la fiul potrivit al lui x.

   a. Dacă $c_i[x]$ are numai $t-1$ chei, dar are un nod frate în stânga sau în dreapta lui care

are t chei, are loc mutarea unei chei din x în $c_i[x]$, apoi mutarea unei chei în x din fratele din stânga sau dreapta a lui $c_i[x]$.

b. Dacă $c_i[x]$ şi fraţii lui au câte $t-1$ chei, fuzionează c_i cu unul dintre fraţi, ceea ce implică mutarea unei chei din x în nodul nou fuzionat ca şi cheie mediană.



Deoarece majoritatea cheilor dintr-un B-arbore sunt în frunze, ne aşteptăm ca, în practică, cele mai multe ştergeri să aibă loc în frunze. Procedura de ştergere ŞTERGE-B-ARBORE acţionează descendent şi fără reveniri. Când se şterge o cheie dintr-un nod intern, au loc o serie de înlocuiri de chei, astfel încât ştergerea efectivă să se facă în frunză (cazurile 2a sau 2b).

Deşi procedura pare complicată, ea presupune numai $O(h)$ accese la disc pentru un B-arbore de înălţime h, deoarece între două apeluri recursive au loc numai $O(1)$ accese la disc. Timpul CPU necesar este de $O(th) = O(t \log_t n)$.

Exerciţii

19.3-1 Prezentaţi rezultatele obţinute după ştergerea, în această ordine, a cheilor C, P şi V din B-arborele din figura 19.8(f).

19.3-2 Descrieţi, în pseudocod, procedura ŞTERGE-B-ARBORE.

Probleme

19-1 Stive în memoria secundară

Considerăm implementarea unei stive pentru un sistem de calcul care are o memorie internă rapidă relativ mică şi o memorie secundară pe disc lentă, dar practic oricât de mare. Operaţiile specifice PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ au ca operand un cuvânt. Se admite creşterea oricât de mare a acestei stive; ceea ce depăşeşte memoria internă, se extinde automat pe disc.

O implementare simplă, dar ineficientă, păstrează întreaga stivă pe disc. În memorie se păstrează doar un pointer la stivă, care este adresa elementului din vârful stivei. Dacă p este valoarea acestui pointer, m este numărul de cuvinte dintr-o pagină, atunci $\lfloor p/m \rfloor$ este numărul paginii de pe disc, iar primul element este al p mod m-lea cuvânt din pagină.

Pentru implementarea procedurii PUNE-ÎN-STIVĂ, se incrementează pointerul de stivă, se citeşte pagina disc corespunzătoare, se copiază conţinutul cuvântului în vârful paginii din vârful stivei, după care pagina se rescrie pe disc. Operaţia SCOATE-DIN-STIVĂ este similară: se decrementează pointerul de stivă, se citeşte pagina corespunzătoare, se extrage elementul din vârful stivei după care pagina se rescrie pe disc.

Deoarece operaţiile cu discul sunt relativ costisitoare, vom folosi numărul de operaţii disc ca măsură a eficienţei. Vom contoriza timpul CPU şi vom cere $\Theta(m)$ pentru fiecare acces disc la o pagină de m cuvinte.

a. Care este valoarea asimptotică a numărului de accese disc la o stivă cu n operaţii asupra stivei, în cel mai defavorabil caz? Care este timpul CPU pentru n operaţii pe stivă? (Răspunsul se va exprima în funcţie de m şi n).

Să considerăm o implementare în care se păstrează permanent în memoria internă o pagină a stivei. (Evident, vom şti copia cărei pagini disc se află în memorie). Se pot face operaţii asupra stivei numai dacă pagina respectivă se află în memorie. Dacă este cazul, pagina curentă de pe disc se rescrie şi se reciteşte alta. Dacă pagina este deja în memorie, nu se mai accesează discul pentru ea.

b. Care este, în cel mai defavorabil caz, numărul de accese la disc şi timpul CPU pentru n operaţii PUNE-ÎN-STIVĂ?

c. Care este, în cel mai defavorabil caz, numărul de accese la disc şi timpul CPU pentru n operaţii asupra stivei?

Presupunem că realizăm o implementare a stivei în care se păstrează două pagini disc în memorie (cu gestiunea aferentă lor).

d. Descrieţi cum se gestionează, în acest caz, paginile stivei, astfel încât numărul amortizat al acceselor de disc să fie $O(1/m)$, iar timpul mediu amortizat CPU să fie $O(1)$.

19-2 Fuzionarea şi divizarea 2-3-4 arborilor

Operaţia de ***fuzionare*** necesită două mulţimi dinamice S' şi S'' şi un element x, astfel încât, pentru orice $x' \in S'$ şi $x'' \in S''$, avem $cheie[x'] < cheie[x] < cheie[x'']$. În urma operaţiei, se returnează o mulţime $S = S' \cup \{x\} \cup S''$. Operaţia de ***divizare*** este oarecum "inversă" fuziunii: se dă o mulţime dinamică S şi un element $x \in S$ şi se creează o mulţime S' care conţine toate elementele din $S - \{x\}$ cu cheile mai mici decât $cheie[x]$ şi o mulţime S'' cu elementele din $S - \{x\}$ care au cheile mai mari decât $cheie[x]$. În acest sens vom examina cum se pot implementa aceste operaţii la 2-3-4 arbori. Pentru simplificare, presupunem că toate cheile sunt distincte.

a. Arătaţi cum se întreţine un câmp *înălţime*$[x]$ care, pentru un nod x al unui 2-3-4 arbore, reprezintă înălţimea subarborelui de rădăcină x. Desigur, soluţia nu trebuie să afecteze timpii asimptotici de execuţie ai operaţiilor de căutare, inserare şi ştergere.

b. Arătaţi cum se implementează operaţia de fuzionare. Fiind daţi 2-3-4 arborii T' şi T'' şi o cheie k, operaţia de fuziune trebuie să se execute în timp $O(|h' - h''|)$, unde h' şi h'' sunt înălţimile arborilor T' şi, respectiv, T''.

c. Într-un 2-3-4 arbore T, considerăm un drum p de la rădăcină la o cheie k, mulţimea S' formată din cheile din T mai mici decât k şi mulţimea S'' formată din cheile din T mai mari decât k. Arătaţi că p împarte mulţimea S' într-o mulţime de arbori $\{T'_0, T'_1, \ldots T'_m\}$ şi o mulţime de chei $\{k'_1, k'_2, \ldots, k'_m\}$, unde, pentru $i = 1, 2, \ldots, m$, avem $y < k'_i < z$, pentru orice cheie $y \in T'_{i-1}$ şi $z \in T'_i$. Care este relaţia dintre înălţimile arborilor T'_{i-1} şi T'_i? Descrieţi cum împarte p mulţimea S'' în submulţimi de arbori şi chei.

d. Arătaţi cum se poate implementa operaţia de diviziune în T. Folosiţi operaţia de fuziune pentru a asambla cheile din S' într-un singur 2-3-4 arbore T' şi cheile din S'' într-un singur 2-3-4 arbore T''. Timpul de execuţie trebuie să fie $O(\lg n)$, unde n este numărul de chei din T. (*Indicaţie:* Costurile reunirii ar trebui să telescopeze.)

Note bibliografice

Knuth [123], Aho, Hopcroft şi Ullman [4], Sedgewick [175] au tratat diverse aspecte ale schemelor de arbori echilibraţi şi B-arbori. O sinteză exhaustivă asupra B-arborilor a fost realizată de Comer [48]. Guibas şi Sedgewick [93] au tratat relaţiile între diversele scheme de arbori echilibraţi, inclusiv arbori roşu-negru şi 2-3-4 arbori.

În 1970, J. E. Hopcroft a inventat 2-3 arborii, precursori ai B-arborilor şi 2-3-4 arborilor, în care fiecare nod intern are doi sau trei fii. B-arborii au fost definiţi de Bayer şi McCreight în 1972 [18]; ei nu au explicat sursa alegerii acestui nume.

20 Heap-uri binomiale

Acest capitol şi capitolul 21 prezintă structuri de date cunoscute sub numele de ***heap-uri interclasabile***, caracterizate de următoarele cinci operaţii.

CREEAZĂ-HEAP() creează şi returnează un heap nou care nu conţine elemente.

INSEREAZĂ(H, x) inserează nodul x, a cărui *cheie* a fost iniţializată, în heap-ul H.

MINIMUM(H) returnează un pointer la nodul cu cea mai mică cheie din H.

EXTRAGE-MIN(H) şterge nodul cu cheia minimă din H şi returnează un pointer la acest nod.

REUNEŞTE(H_1, H_2) creează şi returnează un heap nou care conţine toate nodurile heap-urilor H_1 şi H_2. Heap-urile H_1 şi H_2 sunt "distruse" în urma acestei operaţii.

În plus, pe structurile de date din aceste capitole se pot defini şi următoarele două operaţii.

DESCREŞTE-CHEIE(H, x, k) atribuie nodului x din heap-ul H valoarea k pentru cheie, valoare presupusă a nu fi mai mare decât valoarea curentă a cheii.

ŞTERGE(H, x) şterge nodul x din heap-ul H.

Tabelul din figura 20.1 arată că heap-urile binare obişnuite, folosite, de exemplu, în heapsort (capitolul 7), suportă bine aceste operaţii, cu excepţia operaţiei REUNEŞTE. În cazul heap-urilor binare, pentru fiecare operaţie, exceptând REUNEŞTE, timpul de execuţie în cazul cel mai defavorabil este $O(\lg n)$ (sau mai bun). Dacă totuşi un heap binar trebuie să execute operaţia REUNEŞTE aceasta va fi lentă. Prin concatenarea celor două tablouri care memorează heap-urile binare care se interclasează şi apoi aplicarea operaţiei RECONSTITUIE-HEAP, timpul de execuţie, în cazul cel mai defavorabil, pentru operaţia REUNEŞTE este $\Theta(n)$.

În acest capitol vom examina "heap-urile binomiale" a căror margini pentru timpii de execuţie, în cazurile cele mai defavorabile, sunt prezentate în figura 20.1. În particular, operaţia REUNEŞTE pentru interclasarea a două heap-uri binomiale cu n elemente va fi de complexitate $O(\lg n)$.

În capitolul 21 vom examina heap-urile Fibonacci care pentru anumite operaţii au estimări chiar mai bune a timpilor de execuţie. Să notăm aici că estimările din figura 20.1 pentru heap-urile Fibonacci sunt amortizate şi nu reprezintă margini ale timpilor de execuţie, în cazurile cele mai defavorabile.

În acest capitol vor fi ignorate operaţiile de alocare a nodurilor înainte de o inserare şi de eliberare a nodurilor după o ştergere. Presupunem că de aceste detalii este responsabil codul care apelează operaţiile heap-ului.

Heap-urile binare, binomiale şi Fibonacci sunt ineficiente în raport cu operaţia de CĂUTARE; pentru găsirea unui nod care conţine o anumită valoare nu se poate stabili o cale de căutare directă în aceste structuri. Din acest motiv, operaţiile DESCREŞTE-CHEIE şi ŞTERGE care se referă la un anumit nod reclamă ca parametru de intrare un pointer la nodul respectiv. În cele mai multe aplicaţii această cerinţă nu ridică probleme.

În secţiunea 20.1 se definesc heap-urile binomiale, după ce în prealabil se definesc arborii binomiali. Se introduce, de asemenea, şi o reprezentare particulară a heap-urilor binomiale. În secţiunea 20.2 se prezintă modul în care se pot implementa operaţiile pe heap-uri binomiale pentru a obţine estimările de timp din figura 20.1.

Procedură	Heap binar (cazul cel mai defavorabil)	Heap binomial (cazul cel mai defavorabil)	Heap Fibonacci (amortizat)
CREEAZĂ-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSEREAZĂ	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRAGE-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
REUNEŞTE	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DESCREŞTE-CHEIE	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
ŞTERGE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Figura 20.1 Timpi de execuţie a operaţiilor pentru trei implementări ale heap-urilor interclasabile. Numărul elementelor din heap-ul sau heap-urile folosite de o operaţie este notat cu n.

20.1. Arbori binomiali şi heap-uri binomiale

Deoarece un heap binomial este o colecţie de arbori binomiali, această secţiune începe prin definirea arborilor binomiali şi demonstrarea unor proprietăţi esenţiale ale acestora. Vom defini apoi heap-urile binomiale şi vom arăta cum pot fi reprezentate acestea.

20.1.1. Arbori binomiali

Arborele binomial B_k este un arbore ordonat definit recursiv (a se vedea secţiunea 5.5.2). După cum arată figura 20.2(a), arborele binomial B_0 constă dintr-un singur nod. Arborele binomial B_k constă din doi arbori binomiali B_{k-1} care sunt ***înlănţuiţi***: rădăcina unuia dintre ei este fiul situat cel mai în stânga rădăcinii celuilalt arbore. Figura 20.2(b) prezintă arborii binomiali de la B_0 la B_4.

Lema următoare conţine câteva proprietăţi ale arborilor binomiali.

Lema 20.1 **(Proprietăţi ale arborilor binomiali)** Arborele binomial B_k are:

1. 2^k noduri,
2. înălţimea k,
3. exact $\binom{k}{i}$ noduri, la adâncimea i, pentru $i = 0, 1, \ldots, k$,
4. rădăcina de grad k, grad mai mare decât a oricărui alt nod; mai mult, dacă fiii rădăcinii sunt numerotaţi de la stânga spre dreapta prin $k-1$, $k-2$, ..., 0, atunci fiul i este rădăcina subarborelui B_i.

Demonstraţie. Demonstraţia este inductivă în raport cu k. Pasul iniţial pentru fiecare proprietate îl constituie arborele binomial B_0. Verificarea proprietăţilor pentru B_0 este imediată.

Presupunem că lema are loc pentru arborele B_{k-1}.

1. Arborele binomial B_k are $2^{k-1} + 2^{k-1} = 2^k$ noduri, deoarece el constă din două copii ale arborelui B_{k-1}.

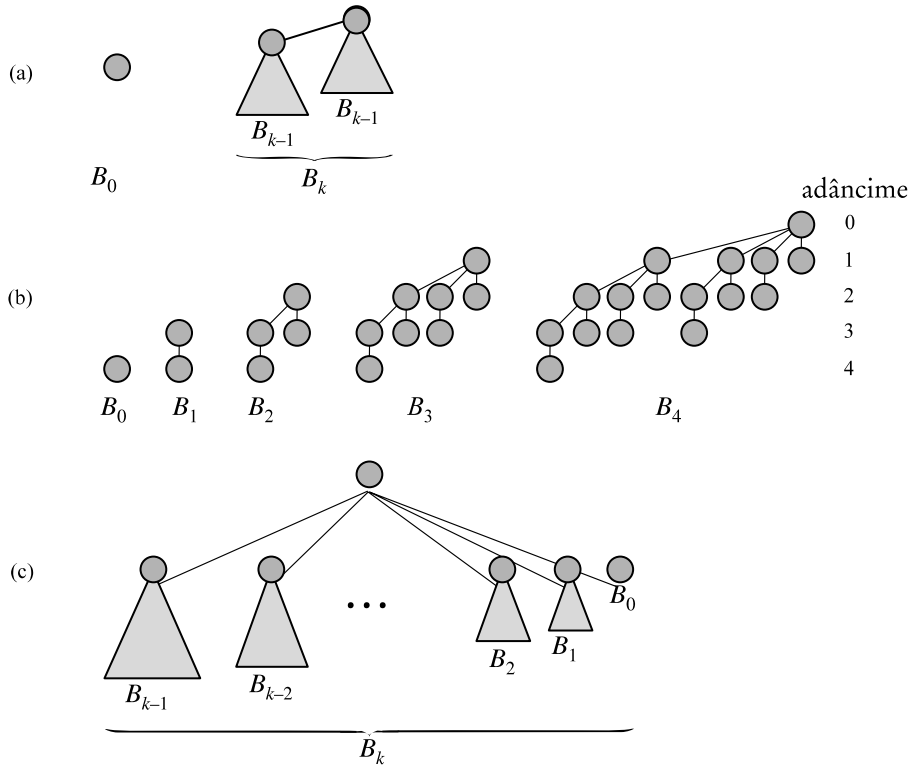


Figura 20.2 **(a)** Definiţia recursivă a arborelui binomial B_k. Triunghiurile reprezintă subarborii rădăcinii. **(b)** Arborii binomiali B_0 până la B_4. Este redată şi adâncimea nodurilor lui B_4. **(c)** Un alt mod de a privi arborele binomial B_k.

2. Ţinând cont de modul în care sunt înlănţuite cele două copii ale lui B_{k-1} pentru a forma B_k, rezultă că adâncimea maximă a lui B_k este cu unu mai mare decât adâncimea maximă a lui B_{k-1}. Din ipoteza de inducţie obţinem valoarea $(k-1)+1=k$ pentru adâncimea maximă a lui B_k.

3. Fie $D(k,i)$ numărul nodurilor situate pe nivelurile i al arborelui binomial B_k. Deoarece B_k este compus din două copii înlănţuite ale lui B_{k-1}, un nod de pe nivelul i din B_{k-1} apare în B_k pe nivelele i şi $i+1$. Altfel spus, numărul nodurilor situate pe nivelul i în B_k este egal cu numărul nodurilor de pe nivelul i din B_{k-1} plus numărul nodurilor de pe nivelul $i-1$ din B_{k-1}. Astfel,

$$D(k,i) = D(k-1,i) + D(k-1,i-1) = \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}.$$

   A doua egalitate rezultă din ipoteza de inducţie, iar egalitatea a treia rezultă din exerciţiul 6.1-7.

4. Singurul nod cu grad mai mare în B_k decât în B_{k-1} este rădăcina, care are cu un fiu mai mult decât în B_{k-1}. Deoarece rădăcina lui B_{k-1} are gradul $k-1$, rădăcina lui B_k are

gradul k. Din ipoteza de inducţie şi după cum ilustrează figura 20.2(c), fiii lui B_{k-1} (de la stânga la dreapta) sunt rădăcinile arborilor B_{k-2}, B_{k-3}, ..., B_0. Când B_{k-1} este legat la B_{k-1}, fiii rădăcinii rezultate sunt rădăcini pentru B_{k-1}, B_{k-2}, ..., B_0. ∎

Corolarul 20.2 Gradul maxim al nodurilor unui arbore binomial având n noduri este $\lg n$.

Demonstraţie. Rezultă imediat din proprietăţile 1 şi 4 ale lemei 20.1. ∎

Proprietatea 3 a lemei 20.1 justifică termenul de "arbore binomial" prin faptul că $\binom{k}{i}$ sunt coeficienţii binomiali. O justificare în plus a acestui termen este oferită de exerciţiul 20.1-3.

20.1.2. Heap-uri binomiale

Un ***heap binomial*** H este format dintr-o mulţime de arbori binomiali care satisfac următoarele ***proprietăţi de heap binomial***.

1. Fiecare arbore binomial din H satisface ***proprietatea de ordonare a unui heap***: cheia unui nod este mai mare sau egală decât cheia părintelui său.

2. Există cel mult un arbore binomial în H a cărui rădăcină are un grad dat.

Conform primei proprietăţi, rădăcina unui arbore cu proprietatea de heap ordonat, conţine cea mai mică cheie din arbore.

Proprietatea a doua implică faptul că un heap binomial H având n noduri conţine cel mult $\lfloor \lg n \rfloor + 1$ arbori binomiali. Pentru o justificare a acestei afirmaţii observaţi că reprezentarea binară a lui n are $\lfloor \lg n \rfloor + 1$ biţi, fie aceştia $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \ldots, b_0 \rangle$, astfel încât $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$. Din proprietatea 1 a lemei 20.1 rezultă că arborele binomial B_i apare în H dacă şi numai dacă bitul $b_i = 1$. Astfel, heap-ul binomial H conţine cel mult $\lfloor \lg n \rfloor + 1$ arbori binomiali.

Figura 20.3(a) prezintă un heap binomial H având 13 noduri. Reprezentarea binară a numărului 13 este $\langle 1101 \rangle$, iar H conţine arborii binomiali cu proprietatea de heap B_3, B_2 şi B_0, având 8, 4 şi respectiv 1 nod, în total fiind 13 noduri.

Reprezentarea heap-urilor binomiale

După cum ilustrează şi figura 20.3(b), fiecare arbore binomial al unui heap binomial este memorat conform reprezentării stânga-fiu, dreapta-frate prezentată în secţiunea 11.4. Fiecare nod are un câmp *cheie* plus alte informaţii specifice aplicaţiei care foloseşte heap-ul. În plus, fiecare nod x conţine pointerii $p[x]$ spre părintele lui, ***fiu***$[x]$ spre fiul situat cel mai în stânga şi ***frate***$[x]$ spre fratele lui x situat imediat în dreapta. Dacă nodul x este o rădăcină atunci $p[x] = \text{NIL}$. Dacă nodul x nu are fii atunci ***fiu***$[x] = \text{NIL}$, iar dacă x este fiul situat cel mai în dreapta, atunci ***frate***$[x] = \text{NIL}$. Fiecare nod conţine de asemenea câmpul ***grad***$[x]$, care reprezintă numărul fiilor lui x.

Rezultă din figura 20.3 că rădăcinile arborilor binomiali conţinuţi de un heap binomial sunt păstrate într-o listă înlănţuită pe care o vom numi în continuare ***listă de rădăcini***. La o traversare a listei de rădăcini, gradul rădăcinilor formează un şir strict crescător. Din a doua proprietate de heap binomial, rezultă că gradele rădăcinilor unui heap binomial având n noduri formează o submulţime a mulţimii $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$. Câmpul ***frate*** are semnificaţii diferite după

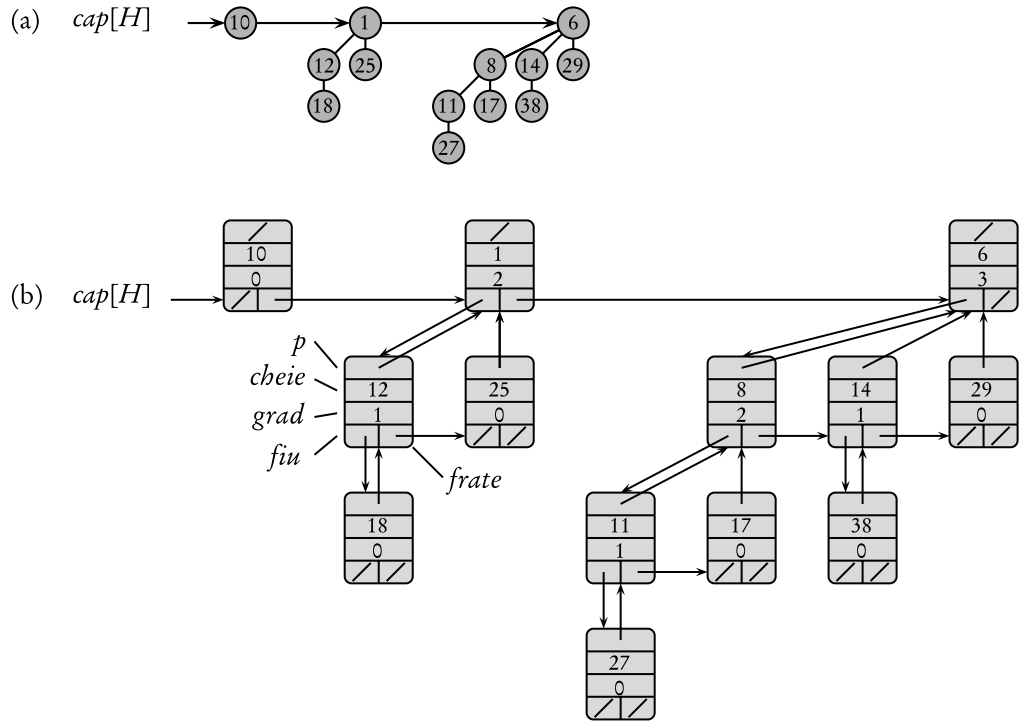


Figura 20.3 Un heap binomial H având $n = 13$ noduri. **(a)** Heap-ul constă din arborii binomiali B_0, B_2 şi B_3, arbori care au 1, 4 şi respectiv 8 noduri, în total fiind $n = 13$ noduri. Deoarece fiecare arbore binomial satisface proprietatea de ordonare a unui heap, cheia fiecărui nod nu este mai mică decât cheia părintelui său. Este redată de asemenea şi lista de rădăcini, care este o listă înlănţuită conţinând rădăcinile în ordine crescătoare a gradelor lor. **(b)** O reprezentare mai detaliată a heap-ului binomial H. Fiecare arbore binomial este memorat conform reprezentării stânga-fiu, dreapta-frate. Fiecare nod memorează de asemenea şi gradul lui.

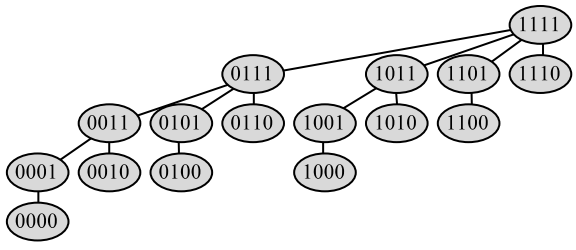


Figura 20.4 Arborele binomial B_4 cu nodurile etichetate prin secvenţe binare corespunzătoare parcurgerii în postordine.

cum nodurile sunt rădăcini sau nu. Dacă x este rădăcină atunci $\mathit{frate}[x]$ referă rădăcina următoare în lista de rădăcini. (Dacă x este ultima rădăcină din listă atunci, ca de obicei, $\mathit{frate}[x] = \text{NIL}$.)

Un heap binomial dat H este referit prin pointerul $\mathit{cap}[H]$ spre prima rădăcină din lista de

rădăcini a lui H. Dacă heap-ul binomial H nu are elemente, atunci $cap[H] = \text{NIL}$.

Exerciţii

20.1-1 Presupunem că x este un nod al unui arbore binomial dintr-un heap binomial şi că $frate[x] \neq \text{NIL}$. Ce relaţie există între $grad[frate[x]]$ şi $grad[x]$, dacă x nu este o rădăcină? Dar dacă x este o rădăcină?

20.1-2 Dacă x nu este nodul rădăcină al unui arbore binomial dintr-un heap binomial, ce relaţie există între $grad[p[x]]$ şi $grad[x]$?

20.1-3 Presupunem că etichetăm nodurile arborelui binomial B_k prin secvenţe binare corespunzătoare parcurgerii în postordine a arborelui, conform figurii 20.4. Considerăm un nod x de pe nivelul i etichetat cu l şi fie $j = k - i$. Arătaţi că eticheta asociată nodului x are j cifre de 1. Câte şiruri binare de lungime k conţin exact j cifre binare de 1? Arătaţi că gradul lui x este egal cu numărul cifrelor 1 situate în dreapta celui mai din dreapta 0 din reprezentarea binară a lui l.

20.2. Operaţii pe heap-uri binomiale

În această secţiune vom arăta cum putem efectua operaţiile pe heap-uri binomiale respectând marginile de timp prezentate în figura 20.1. Vom demonstra numai rezultatele privind marginile superioare; justificarea rezultatelor referitoare la marginile inferioare cuprinsă în exerciţiul 20.2-10.

Crearea unui heap binomial nou

Pentru a crea un heap binomial vid, procedura CREEAZĂ-HEAP-BINOMIAL va aloca şi returna un obiect H, pentru care $cap[H] = \text{NIL}$. Timpul de execuţie este $\Theta(1)$.

Găsirea cheii minime

Procedura HEAP-BINOMIAL-MIN returnează un pointer la nodul cu cea mai mică cheie dintr-un heap binomial H având n noduri. Această implementare presupune că nu există chei cu valoarea ∞. (A se vedea Exerciţiul 20.2-5.)

HEAP-BINOMIAL-MIN(H)

```
1: y ← NIL
2: x ← cap[H]
3: min ← ∞
4: cât timp x ≠ NIL execută
5:    dacă cheie[x] < min atunci
6:       min ← cheie[x]
7:       y ← x
8:    x ← frate[x]
9: returnează y
```

Cheia minimă a unui heap binomial se află într-o rădăcină deoarece este un heap ordonat. Procedura HEAP-BINOMIAL-MIN verifică toate rădăcinile (în număr de cel mult $\lfloor \lg n \rfloor + 1$) şi reţine minimul curent în *min*, respectiv un pointer la acest minim în y. Apelată pentru heap-ul binomial din figura 20.3 procedura va returna un pointer la nodul care conţine cheia 1.

Timpul de execuţie al procedurii HEAP-BINOMIAL-MIN este $O(\lg n)$ deoarece există cel mult $\lfloor \lg n \rfloor + 1$ rădăcini verificate.

Reuniunea a două heap-uri binomiale

Operaţia de reuniune a două heap-uri binomiale este folosită de aproape toate celelalte operaţii rămase. Procedura HEAP-BINOMIAL-REUNEŞTE înlănţuie repetat arborii binomiali care au rădăcini de acelaşi grad. Procedura următoare leagă arborele B_{k-1} având nodul rădăcină y la arborele B_{k-1} având nodul rădăcină z; mai precis, z va fi părintele lui y. Nodul z devine astfel rădăcina unui arbore B_k.

BINOMIAL-LEGĂTURĂ(y, z)

1: $p[y] \leftarrow z$
2: $\mathit{frate}[y] \leftarrow \mathit{fiu}[z]$
3: $\mathit{fiu}[z] \leftarrow y$
4: $\mathit{grad}[z] \leftarrow \mathit{grad}[z] + 1$

Procedura BINOMIAL-LEGĂTURĂ plasează nodul y în capul listei înlănţuite care conţine fiii nodului z într-un timp $O(1)$. Reprezentarea stânga-fiu, dreapta-frate a fiecărui arbore binomial asigură succesul acestei proceduri deoarece fiecare arbore binomial are proprietatea de ordonare a arborelui: fiul cel mai din stânga al rădăcinii unui arbore B_k este rădăcina unui arbore B_{k-1}.

Procedura HEAP-BINOMIAL-REUNEŞTE uneşte două heap-uri binomiale H_1 şi H_2 şi returnează heap-ul rezultat. Pe parcursul efectuării operaţiei, reprezentările heap-urilor H_1 şi H_2 sunt distruse. Procedura foloseşte pe lângă procedura BINOMIAL-LEGĂTURĂ încă o procedură auxiliară ANSAMBLU-BINOMIAL-INTERCLASEAZĂ, care interclasează listele de rădăcini ale heap-urilor H_1 şi H_2 într-o singură listă simplu înlănţuită ordonată crescător după gradul nodurilor. Procedura HEAP-BINOMIAL-INTERCLASEAZĂ, a cărei descriere în pseudocod se cere în exerciţiul 20.2-2, este similară procedurii INTERCLASEAZĂ din secţiunea 1.3.1.

Figura 20.5 prezintă un exemplu pentru care se petrec toate cele patru cazuri indicate prin comentarii în procedura HEAP-BINOMIAL-REUNEŞTE.

Procedura HEAP-BINOMIAL-REUNEŞTE se desfăşoară în două faze. În prima fază se interclasează (prin apelul HEAP-BINOMIAL-INTERCLASEAZĂ) listele de rădăcini ale heap-urilor binomiale H_1 şi H_2 într-o listă simplu înlănţuită H care este ordonată crescător în raport cu gradul nodurilor rădăcină. Lista formată poate conţine cel mult două rădăcini cu acelaşi grad. Astfel, în faza a doua sunt unite toate rădăcinile care au acelaşi grad, astfel încât să nu existe două noduri cu acelaşi grad. Deoarece lista înlănţuită H este ordonată după grad, operaţiile de înlănţuire din faza a doua sunt efectuate rapid.

Detaliem cele două faze ale procedurii. Liniile 1–3 încep prin interclasarea celor două liste ale heap-urilor binomiale H_1 şi H_2 într-o singură listă de rădăcini H. Listele de rădăcini ale lui H_1 şi H_2 sunt ordonate strict crescător după grad, iar HEAP-BINOMIAL-INTERCLASEAZĂ returnează o listă de rădăcini H, ordonată crescător după grad. Dacă listele H_1 şi H_2 au împreună m noduri, atunci timpul de execuţie pentru HEAP-BINOMIAL-INTERCLASEAZĂ este $O(m)$, datorat

examinării repetate a rădăcinilor din capul listelor şi adăugării rădăcinii având gradul mai mic în lista de rădăcini rezultat, eliminând această rădăcină din lista dată la intrare.

HEAP-BINOMIAL-REUNEŞTE(H_1, H_2)
1: $H \leftarrow$ CREEAZĂ-HEAP-BINOMIAL()
2: *cap*$[H] \leftarrow$ HEAP-BINOMIAL-INTERCLASEAZĂ(H_1, H_2)
3: eliberează obiectele H_1 şi H_2 dar nu şi listele referite de ele
4: **dacă** *cap*$(H) =$ NIL **atunci**
5: **returnează** H
6: *prec-x* $\leftarrow$ NIL
7: $x \leftarrow$ *cap*$[H]$
8: *urm-x* $\leftarrow$ *frate*$[x]$
9: **cât timp** *urm-x* $\neq$ NIL **execută**
10: **dacă** (*grad*$[x] \neq$ *grad*[*urm-x*])
sau (*frate*[*urm-x*] $\neq$ NIL şi *grad*$[frate[$*urm-x*$]] =$ *grad*$[x]$) **atunci**
11: *prec-x* $\leftarrow x \triangleright$ Cazurile 1 şi 2
12: $x \leftarrow$ *urm-x* $\triangleright$ Cazurile 1 şi 2
13: **altfel**
14: **dacă** *cheie*$[x] \leq$ *cheie*[*urm-x*] **atunci**
15: *frate*$[x] \leftarrow$ *frate*[*urm-x*] $\triangleright$ Cazul 3
16: BINOMIAL-LEGĂTURĂ(*urm-x*$, x) \triangleright$ Cazul 3
17: **altfel**
18: **dacă** *prec-x* $=$ NIL **atunci**
19: *cap*$[H] \leftarrow$ *urm-x* $\triangleright$ Cazul 4
20: **altfel**
21: *frate*[*prec-x*] $\leftarrow$ *urm-x* $\triangleright$ Cazul 4
22: BINOMIAL-LEGĂTURĂ($x,$ *urm-x*$) \triangleright$ Cazul 4
23: $x \leftarrow$ *urm-x* $\triangleright$ Cazul 4
24: *urm-x* $\leftarrow$ *frate*$[x] \triangleright$ Cazul 4
25: **returnează** H

În continuare procedura HEAP-BINOMIAL-REUNEŞTE iniţializează câţiva pointeri în lista de rădăcini H. Dacă heap-urile binomiale date la intrare sunt vide, atunci în liniile 4–5 se iese din procedură. Începând cu linia 6 ne situăm în cazul în care H conţine cel puţin o rădăcină. Din acest punct se păstrează trei pointeri în lista de rădăcini:

- x indică rădăcina curentă examinată,
- *prec-x* indică rădăcina precedentă lui x în lista de rădăcini: *frate*[*prec-x*] $= x$, şi
- *urm-x* indică rădăcina următoare lui x în listă: *frate*$[x]$ = *urm-x*.

H poate conţine iniţial, cel mult două rădăcini cu un grad dat: deoarece H_1 şi H_2 sunt heap-uri binomiale, ele nu au două rădăcini având acelaşi grad. Mai mult, procedura HEAP-BINOMIAL-INTERCLASEAZĂ ne garantează că dacă H conţine două rădăcini având acelaşi grad, atunci ele sunt adiacente în lista de rădăcini.

În timpul execuţiei procedurii HEAP-BINOMIAL-REUNEŞTE, de fapt, pot exista trei rădăcini având acelaşi grad. Vom vedea imediat când se produce această situaţie. La fiecare iteraţie a

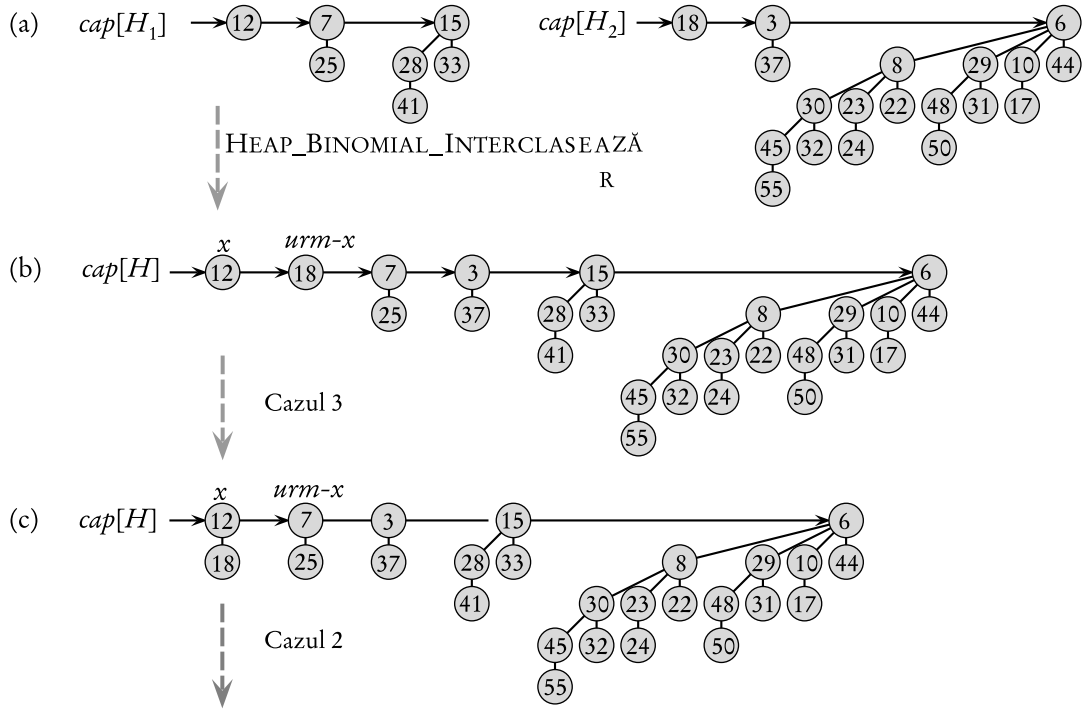


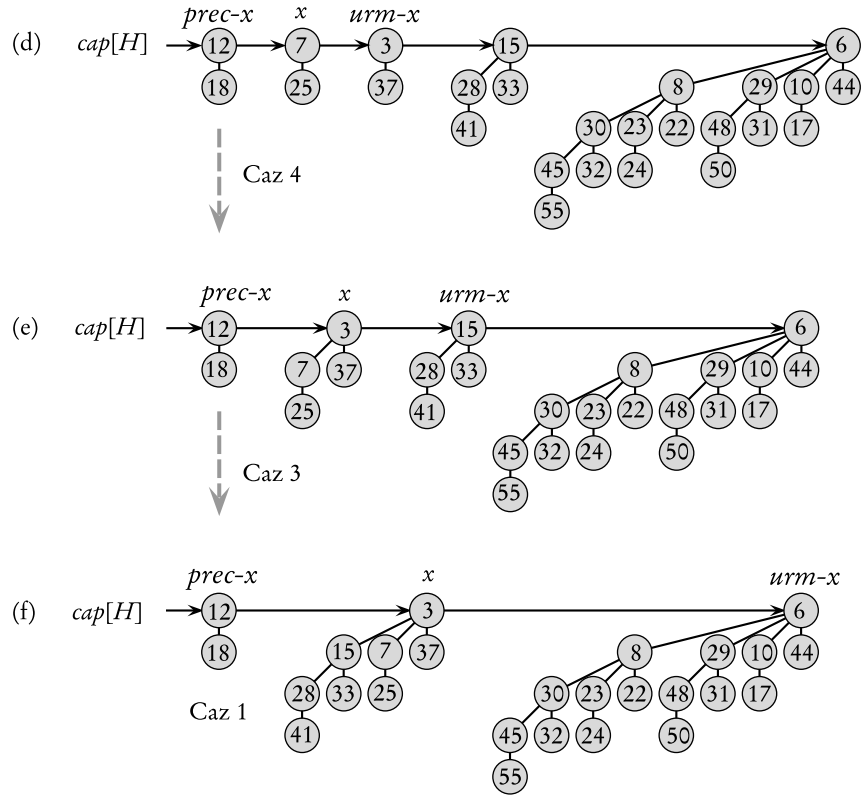
Figura 20.5 Execuţia procedurii HEAP-BINOMIAL-REUNEŞTE. **(a)** Heap-urile binomiale H_1 şi H_2. **(b)** Heap-ul binomial H este rezultatul operaţiei HEAP-BINOMIAL-REUNEŞTE(H_1, H_2). Iniţial x este prima rădăcină din lista de rădăcini a lui H. Se aplică cazul 3, deoarece atât x cât şi *urm-x* au acelaşi grad 0 şi $cheie[x] < cheie[urm\text{-}x]$. **(c)** După realizarea legăturii se aplică cazul 2, deoarece x este prima din cele trei rădăcini cu acelaşi grad. **(d)** După ce toţi pointerii au avansat cu o poziţie în lista de rădăcini, se aplică cazul 4 deoarece x este prima din cele două rădăcini cu grad egal. **(e)** După apariţia legăturii se aplică cazul 3. **(f)** După stabilirea unei noi legături se aplică cazul 1 pentru că x are gradul 3, iar *urm-x* are gradul 4. Aceasta este ultima iteraţie a ciclului **cât timp**, deoarece după deplasarea pointerilor în lista de rădăcini, *urm-x* = NIL.

ciclului **cât timp** din liniile 9–24 se decide dacă se poate lega x şi *urm-x* în funcţie de gradul lor şi de gradul lui *frate*[*urm-x*]. Un invariant al acestui ciclu este faptul că la fiecare reluare a corpului ciclului atât x cât şi *urm-x* sunt diferiţi de NIL.

Cazul 1, prezentat în figura 20.6(a), se produce atunci când $grad[x] \neq$ *grad*[*urm-x*], adică x este rădăcina unui arbore B_k şi *urm-x* este rădăcina unui arbore B_l pentru un $l > k$. Această situaţie este tratată în liniile 11–12. Deoarece nu trebuie să legăm x şi *urm-x*, nu rămâne decât să deplasăm pointerii în listă. Actualizarea pointerului *urm-x* pentru a referi nodul ce urmează noului nod x este efectuată în linia 24, deoarece aceasta este comună tuturor cazurilor.

Cazul 2, prezentat în figura 20.6(b), are loc atunci când x este prima rădăcină din cele trei care au acelaşi grad, adică atunci când *grad*$[x]$ = *grad*[*urm-x*] = *grad*[*frate*[*urm-x*]].

Acest caz este tratat similar cu cazul 1: efectuăm doar o deplasare a pointerilor în listă. Testul din linia 10 este comun cazurilor 1 şi 2, la fel cum liniile 11–12 tratează amândouă cazurile.



Cazurile 3 şi 4 se produc atunci când x este prima rădăcină din două rădăcini succesive având acelaşi grad, adică

$$\mathit{grad}[x] = \mathit{grad}[\mathit{urm\text{-}x}] \neq \mathit{grad}[frate[\mathit{urm\text{-}x}]].$$

Aceste cazuri apar la iteraţia următoare după fiecare caz, dar unul din ele urmează imediat după cazul 2. În cazurile 3 şi 4 vom înlănţui x şi *urm-x*. Aceste cazuri diferă între ele după cum x sau *urm-x* au cheia mai mică, fapt ce determină care din noduri va fi rădăcină în procesul de legare a lor.

În cazul 3, prezentat în figura 20.6(c), $\mathit{cheie}[x] \leq \mathit{cheie}[\mathit{urm\text{-}x}]$, astfel că *urm-x* va fi legat la x. Linia 15 şterge *urm-x* din lista de rădăcini, iar în linia 16 *urm-x* devine fiul situat cel mai în stânga lui x.

În cazul 4, prezentat în figura 20.6(d), cheia mai mică o are *urm-x*, deci x este legat la *urm-x*. Liniile 17–21 şterg x din lista de rădăcini. Există două subcazuri, după cum x este (linia 19) sau nu (linia 21) prima rădăcină din listă. În linia 22, x devine fiul situat cel mai în stânga lui *urm-x*, iar linia 23 actualizează x pentru iteraţia următoare.

Pregătirea iteraţiei următoare a ciclului **cât timp** este aceeaşi pentru ambele cazuri 3 şi 4. x referă un arbore B_{k+1} obţinut prin legarea a doi arbori B_k. După operaţia HEAP-BINOMIAL-INTERCLASEAZĂ în lista de rădăcini existau zero, unul sau doi arbori B_{k+1}, deci x este acum prima rădăcină din lista de rădăcini pentru un număr de unu, doi sau trei arbori B_{k+1}. În cazul existenţei unui singur arbore (x referindu-l pe acesta), la iteraţia următoare se va produce cazul 1:

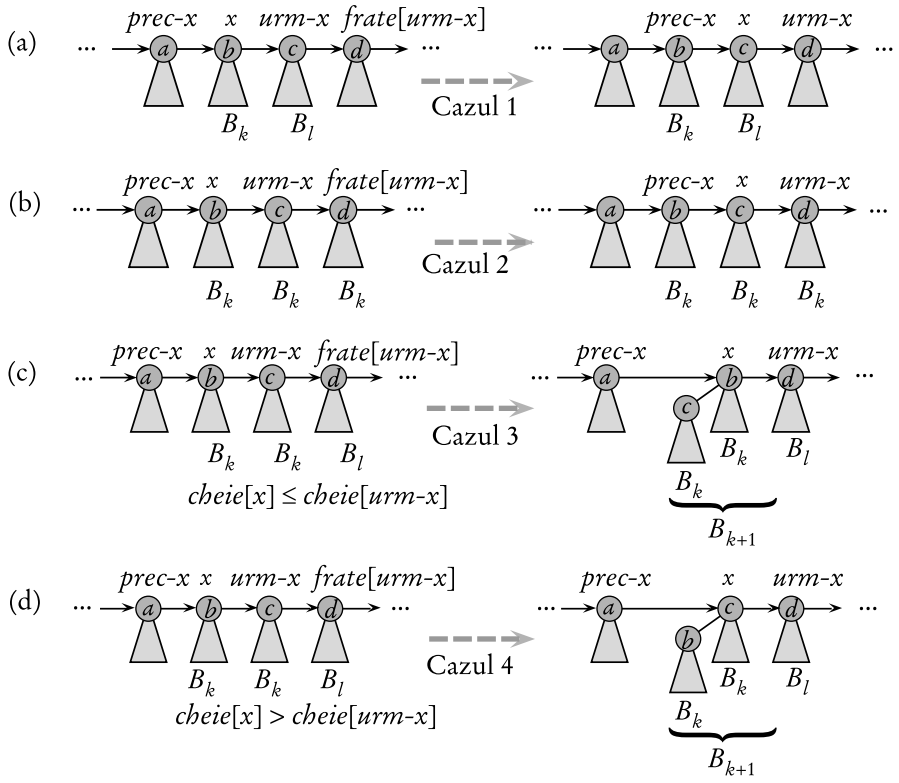


Figura 20.6 Cele patru cazuri care se pot produce în HEAP-BINOMIAL-REUNEŞTE. Etichetele a, b, c şi d servesc aici doar la identificarea rădăcinilor; ele nu reprezintă gradele sau cheile acestor rădăcini. În fiecare din cazuri, x este rădăcina unui arbore B_k şi $l > k$. **(a)** Cazul 1: $grad[x] \neq grad[urm\text{-}x]$. Pointerii se deplasează în lista de rădăcini cu o poziţie în jos. **(b)** Cazul 2: $grad[x] = grad[urm\text{-}x] = grad[frate[urm\text{-}x]]$. Din nou, pointerii se deplasează în lista de rădăcini cu o poziţie spre dreapta şi la iteraţia următoare se execută unul din cazurile 3 sau 4. **(c)** Cazul 3: $grad[x] = grad[urm\text{-}x] \neq grad[frate[urm\text{-}x]]$ şi $cheie[x] \leq cheie[urm\text{-}x]$. Ştergem $urm\text{-}x$ din lista de rădăcini şi îl legăm de x, creând astfel un arbore B_{k+1}. **(d)** Cazul 4: $grad[x] = grad[urm\text{-}x] \neq grad[frate[urm\text{-}x]]$ şi $cheie[urm\text{-}x] \leq cheie[x]$. Ştergem x din lista de rădăcini şi îl legăm la $urm\text{-}x$, creând astfel din nou un arbore B_{k+1}.

grad$[x] \neq$ *grad*[*urm-x*]. Dacă x referă primul arbore din doi existenţi atunci la iteraţia următoare are loc unul din cazurile 3 sau 4. În sfârşit, dacă x referă primul arbore din trei existenţi atunci la iteraţia următoare are loc cazul 2.

Timpul de execuţie pentru HEAP-BINOMIAL-REUNEŞTE este $O(\lg n)$, unde n este numărul total de noduri din heap-urile binomiale H_1 şi H_2. Justificăm acest rezultat după cum urmează. Fie n_1 şi n_2 numărul nodurilor heap-urilor H_1, respectiv H_2 astfel încât $n = n_1 + n_2$. Atunci numărul maxim de rădăcini conţinute de H_1 şi H_2 este $\lfloor \lg n_1 \rfloor + 1$, respectiv $\lfloor \lg n_2 \rfloor + 1$. Astfel, imediat după apelul HEAP-BINOMIAL-INTERCLASEAZĂ, H conţine cel mult $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq 2\lfloor \lg n \rfloor + 2 = O(\lg n)$ rădăcini. Rezultă că timpul de execuţie pentru HEAP-BINOMIAL-INTERCLASEAZĂ este $O(\lg n)$. Fiecare iteraţie a ciclului **cât timp** se execută într-un timp $O(1)$ şi pot exista cel mult $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ iteraţii deoarece de la fiecare iteraţie fie pointerii avansează cu o poziţie în lista H, fie se elimină o rădăcină din lista de rădăcini. Astfel, timpul total de execuţie este $O(\lg n)$.

Inserarea unui nod

Procedura următoare inserează nodul x în heap-ul binomial H. Se presupune că nodul x este creat şi câmpul *cheie*$[x]$ este iniţializat.

HEAP-BINOMIAL-INSEREAZĂ(H, x)
1: $H' \leftarrow$ CREEAZĂ-HEAP-BINOMIAL()
2: $p[x] \leftarrow$ NIL
3: *fiu*$[x] \leftarrow$ NIL
4: *frate*$[x] \leftarrow$ NIL
5: *grad*$[x] \leftarrow 0$
6: *cap*$[H'] \leftarrow x$
7: $H \leftarrow$ HEAP-BINOMIAL-REUNEŞTE(H, H')

Procedura creează un heap binomial H' cu un nod într-un timp $O(1)$ pe care îl reuneşte apoi cu heap-ul binomial H având n noduri într-un timp $O(\lg n)$. Procedura HEAP-BINOMIAL-REUNEŞTE eliberează spaţiul alocat heap-ului binomial temporar H'. (O implementare directă care nu foloseşte HEAP-BINOMIAL-REUNEŞTE se cere în exerciţiul 20.2-8.)

Extragerea nodului având cheia minimă

Procedura următoare extrage nodul având cheia minimă din heap-ul binomial H şi returnează un pointer la nodul extras.

HEAP-BINOMIAL-EXTRAGE-MIN(H)
1: caută rădăcina x cu cheia minimă în lista de rădăcini şi şterge x din lista de rădăcini a lui H
2: $H' \leftarrow$ CREEAZĂ-HEAP-BINOMIAL()
3: inversează ordinea memorării fiilor lui x în lista înlănţuită asociată şi atribuie lui *cap*$[H']$ capul listei rezultate
4: $H \leftarrow$ HEAP-BINOMIAL-REUNEŞTE(H, H')
5: **returnează** x

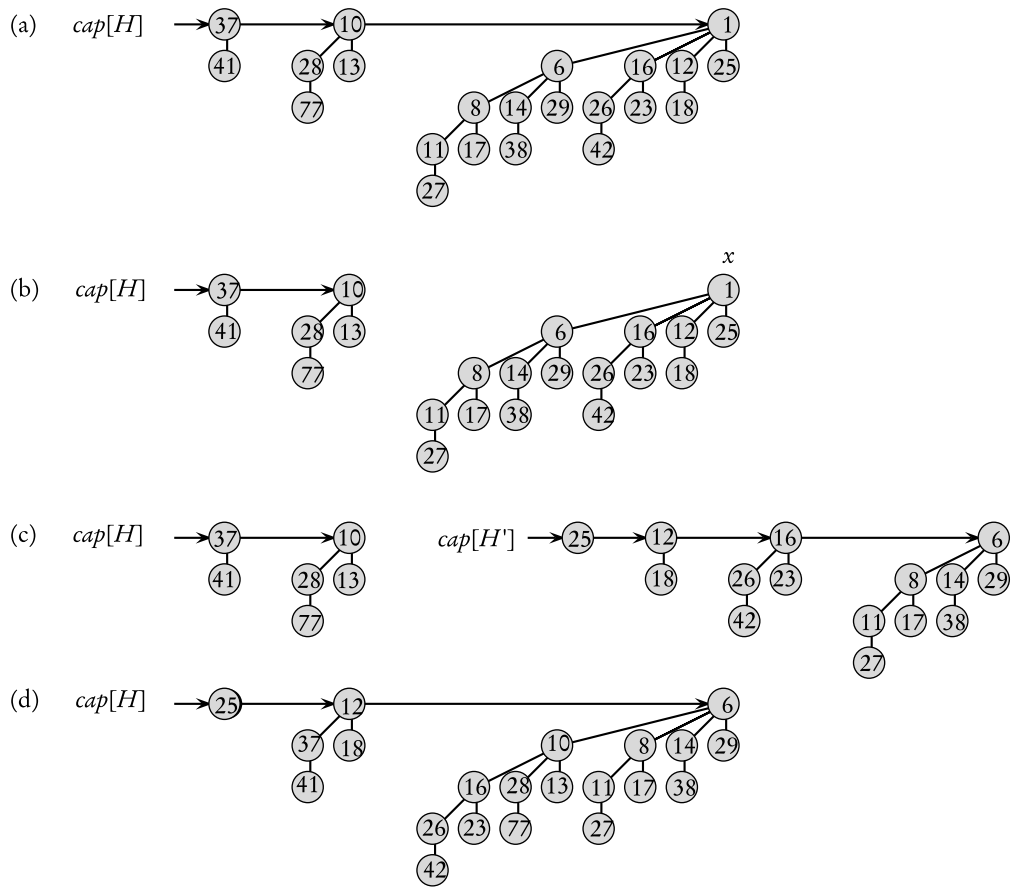


Figura 20.7 Acţiunile procedurii HEAP-BINOMIAL-EXTRAGE-MIN. **(a)** Un heap binomial H. **(b)** Rădăcina x având cheia minimă este eliminată din lista de rădăcini a lui H. **(c)** Lista înlănţuită ce conţine fiii lui x este inversată, obţinându-se un alt heap binomial H'. **(d)** Rezultatul reunirii heap-urilor H şi H'.

Modul de funcţionare al procedurii este ilustrat în figura 20.7. Heap-ul binomial H dat ca parametru de intrare este ilustrat în figura 20.7(a). Figura 20.7(b) prezintă situaţia obţinută după linia 1: rădăcina x având cheia minimă a fost eliminată din lista de rădăcini a lui H. Dacă x este rădăcina unui arbore B_k, atunci din proprietatea 4 a lemei 20.1 rezultă că fiii lui x, de la stânga la dreapta, sunt rădăcinile unor arbori B_{k-1}, B_{k-2}, ..., B_0. În figura 20.7(c) se ilustrează faptul că inversând lista fiilor lui x (în linia 3) obţinem un heap binomial H' care conţine toate nodurile din arborele corespunzător lui x, exceptându-l pe x. Deoarece în linia 1 arborele lui x este şters din H, heap-ul binomial rezultat prin reunirea în linia 4 a lui H şi H', conform figurii 20.7(d), va conţine toate nodurile care existau iniţial în H, exceptându-l desigur pe x. În final, în linia 5 se returnează x.

HEAP-BINOMIAL-EXTRAGE-MIN se execută într-un timp $O(\lg n)$ deoarece fiecare din liniile 1–4 se execută într-un timp $O(\lg n)$.

Descreşterea unei chei

Procedura HEAP-BINOMIAL-DESCREŞTE-CHEIE micşorează valoarea cheii nodului x din heap-ul binomial H, atribuindu-i valoarea k. Dacă k este mai mare decât valoarea curentă a cheii lui x, se semnalează o eroare.

HEAP-BINOMIAL-DESCREŞTE-CHEIE(H, x, k)
1: **dacă** $k > \mathit{cheie}[x]$ **atunci**
2: **eroare** "cheia nouă este mai mare decât cheia existentă"
3: $\mathit{cheie}[x] \leftarrow k$
4: $y \leftarrow x$
5: $z \leftarrow p[y]$
6: **cât timp** $z \neq$ NIL şi $\mathit{cheie}[y] < \mathit{cheie}[z]$ **execută**
7: interschimbă $\mathit{cheie}[y] \leftrightarrow \mathit{cheie}[z]$
8: ▷ Dacă y şi z au şi alte câmpuri atunci interschimbă şi aceste câmpuri.
9: $y \leftarrow z$
10: $z \leftarrow p[y]$

Această procedură descreşte o cheie în mod similar cu metoda aplicată pentru un heap binar: prin "ridicarea" cheii în heap, după cum arată figura 20.8. După ce procedura se asigură că noua cheie nu este mai mare decât cea curentă şi apoi actualizează cheia curentă a lui x, urmează un proces de căutare în sus, iar y referă iniţial nodul x. La fiecare iteraţie a ciclului **cât timp**, în liniile 6–10, se compară $\mathit{cheie}[y]$ cu cheia părintelui z a lui y. Dacă y este chiar rădăcina sau $\mathit{cheie}[y] \geq \mathit{cheie}[z]$, atunci arborele binomial este un heap ordonat. În caz contrar nodul y încalcă proprietatea de ordonare pentru heap, astfel cheile lui y şi z se interschimbă, împreună cu alte informaţii, apoi procedura deplasează y şi z cu un nivel mai sus în arbore şi continuă iteraţiile.

Timpul de execuţie HEAP-BINOMIAL-DESCREŞTE-CHEIE este $O(\lg n)$. Numărul maxim de iteraţii pentru ciclul **cât timp** (liniile 6–10) este $\lfloor \lg n \rfloor$ deoarece, din proprietatea 2 a lemei 20.1 rezultă că înălţimea maximă a lui x este $\lfloor \lg n \rfloor$.

Ştergerea unei chei

Ştergerea cheii şi a altor informaţii asociate unui nod x aparţinând unui heap binomial H se poate desfăşura fără dificultate într-un timp $O(\lg n)$. Implementarea următoare presupune că

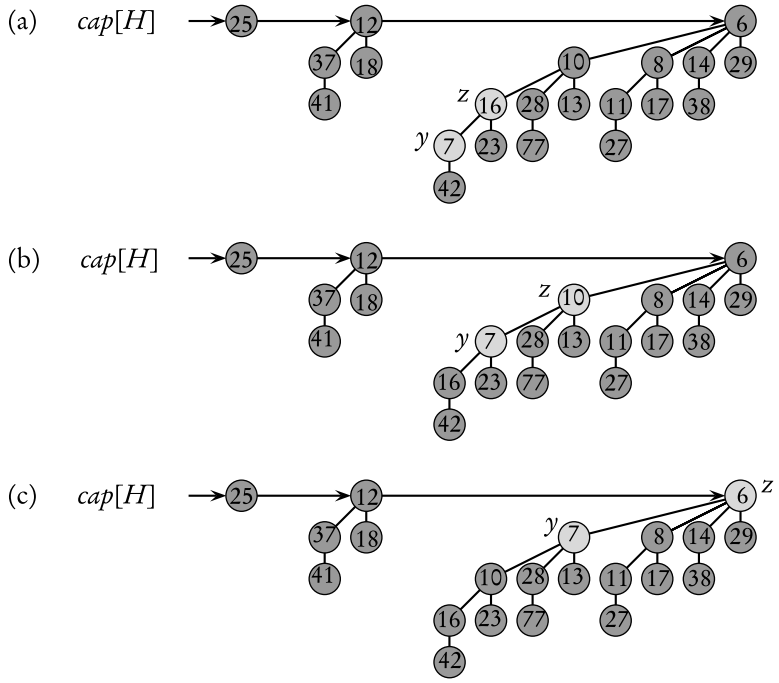


Figura 20.8 Acţiunile procedurii HEAP-BINOMIAL-DESCREŞTE-CHEIE. **(a)** Situaţia înainte de linia 5 la prima iteraţie a ciclului **cât timp**. Cheia nodului y este micşorată la 7, valoare mai mică decât a cheii părintelui z a lui y. **(b)** Cheile celor două noduri sunt interschimbate; este ilustrată situaţia existentă înainte de linia 5 la iteraţia a doua. Pointerii y şi z au fost mutaţi cu un nivel mai sus în arbore, dar încă este încălcată proprietatea de ordonare a heap-ului. **(c)** După o interschimbare şi o deplasare a pointerilor în arbore mai sus cu un nivel, constatăm că proprietatea de ordonare a heap-ului este satisfăcută, deci ciclul **cât timp** se încheie.

nodurile din heap-ul binomial nu au chei având valoarea $-\infty$.

HEAP-BINOMIAL-ŞTERGE(H, x)
1: HEAP-BINOMIAL-DESCREŞTE-CHEIE($H, x, -\infty$)
2: HEAP-BINOMIAL-EXTRAGE-MIN(H)

După ce în procedura HEAP-BINOMIAL-ŞTERGE se atribuie valorea $-\infty$ cheii nodului x, acesta devine nodul având cheia minimă în heap-ul binomial. (Exerciţiul 20.2-6 tratează situaţia în care $-\infty$ nu poate fi valoarea unei chei, nici măcar temporar.) Urmează apoi ca această cheie şi eventual alte informaţii asociate să fie ridicate până la o rădăcină prin apelul HEAP-BINOMIAL-DESCREŞTE-CHEIE. Această rădăcină este apoi eliminată din H prin apelul HEAP-BINOMIAL-EXTRAGE-MIN.

Timpul de execuţie pentru HEAP-BINOMIAL-ŞTERGE este $O(\lg n)$.

Exerciţii

20.2-1 Daţi un exemplu de două heap-uri *binare* fiecare având n elemente, astfel încât CREEAZĂ-HEAP să necesite un timp $\Theta(n)$ pentru a concatena tablourile heap-urilor.

20.2-2 Descrieţi în pseudocod operaţia HEAP-BINOMIAL-INTERCLASEAZĂ.

20.2-3 Determinaţi heap-ul binomial care rezultă după inserarea unui nod având cheia 24 în heap-ul binomial din figura 20.7(d).

20.2-4 Determinaţi heap-ul binomial care rezultă după ştergerea nodului având cheia 28 din heap-ul binomial prezentat în figura 20.8(c).

20.2-5 Explicaţi de ce procedura HEAP-BINOMIAL-MIN nu funcţionează corect în cazul în care pot exista chei având valoarea ∞. Rescrieţi procedura astfel încât să funcţioneze corect şi în aceste cazuri.

20.2-6 Presupunem că nu putem reprezenta valoarea $-\infty$ pentru o cheie. Rescrieţi procedura HEAP-BINOMIAL-ŞTERGE astfel încât să funcţioneze corect în această situaţie. Timpul de execuţie ar trebui păstrat de asemenea $O(\lg n)$.

20.2-7 Examinaţi relaţia între inserarea într-un heap binomial şi incrementarea unui număr binar şi de asemenea, relaţia între reunirea a două heap-uri binomiale şi adunarea a două numere binare.

20.2-8 Luând în considerare exerciţiul 20.2-7, rescrieţi HEAP-BINOMIAL-INSEREAZĂ pentru a insera un nod într-un heap binomial direct, fără a apela HEAP-BINOMIAL-REUNEŞTE.

20.2-9 Arătaţi că dacă listele de rădăcini sunt păstrate în ordine strict descrescătoare după grad (în locul ordinii strict crescătoare), atunci fiecare operaţie a heap-urilor binomiale poate fi implementată fără a modifica timpul de execuţie.

20.2-10 Găsiţi intrări potrivite pentru HEAP-BINOMIAL-EXTRAGE-MIN, HEAP-BINOMIAL-DESCREŞTE-CHEIE şi HEAP-BINOMIAL-ŞTERGE astfel încât acestea să se execute într-un timp $\Omega(\lg n)$. Explicaţi de ce timpii de execuţie în cazurile cele mai defavorabile pentru HEAP-BINOMIAL-INSEREAZĂ, HEAP-BINOMIAL-MIN şi HEAP-BINOMIAL-REUNEŞTE sunt $\overset{\infty}{\Omega}(\lg n)$ şi nu $\Omega(\lg n)$. (A se vedea problema 2-5.)

Probleme

20-1 Heap-uri 2-3-4

Capitolul 19 a introdus arborii 2-3-4, în care fiecare nod intern (nod diferit de rădăcină) are doi, trei sau patru fii şi toate nodurile frunză sunt la aceeaşi adâncime. În această problemă vom implementa ***heap-uri 2-3-4*** pentru care se definesc operaţiile pe heap-uri interclasabile.

Heap-urile 2-3-4 diferă de arborii 2-3-4 după cum urmează. Într-un heap 2-3-4 doar nodurile frunză memorează chei, fiecare frunză x memorând o singură cheie $cheie[x]$. Cheile nu sunt memorate ordonat în frunze; adică, de la stânga spre dreapta cheile pot apărea în orice ordine. Fiecare nod intern x conţine o valoare $mic[x]$ care este egală cu cea mai mică cheie memorată în frunzele subarborelui cu rădăcina x. Rădăcina r conţine un câmp $\mathit{înălţime}[r]$ care reprezintă înălţimea arborelui. În sfârşit, un heap 2-3-4 va fi păstrat în memorie, astfel că operaţiile de citire şi scriere nu sunt necesare.

Implementaţi operaţiile următoare pe heap-uri 2-3-4. Fiecare operaţie din lista (a)–(e) trebuie să se execute pe un heap 2-3-4 având n elemente într-un timp $O(\lg n)$. Operaţia REUNEŞTE specificată la punctul (f) trebuie să se execute într-un timp $O(\lg n)$, unde n este numărul de elemente din heap-urile date ca intrare.

a. MINIMUM – returnează un pointer la frunza cu cea mai mică cheie.

b. DESCREŞTE-CHEIE – descreşte cheia unei frunze date x la o valoare $k \leq \mathit{cheie}[x]$.

c. INSEREAZĂ – care inserează frunza x având cheia k.

d. ŞTERGE – care şterge o frunză dată x.

e. EXTRAGE-MIN – care extrage frunza având cheia cea mai mică.

f. REUNEŞTE – care reuneşte două heap-uri 2-3-4, returnând un singur heap 2-3-4 şi distrugând heap-urile date ca intrare.

20-2 Algoritmul arborelui de acoperire minim folosind heap-uri interclasabile

Capitolul 24 prezintă doi algoritmi pentru găsirea unui arbore de acoperire minim pentru un graf neorientat. Vom vedea cum se pot folosi heap-uri interclasabile pentru a obţine încă un algoritm de determinare a arborelui de acoperire minim.

Fie un graf neorientat $G = (V, E)$ cu o funcţia de cost $w : E \to \mathbb{R}$. Vom numi $w(u, v)$, costul asociat muchiei (u, v). Dorim să găsim un arbore parţial de cost minim pentru G: o submulţime aciclică $T \subseteq E$ care uneşte toate vârfurile din V şi al cărei cost total $w(T)$ este minim, unde

$$w(T) = \sum_{(u,v)\in T} w(u, v).$$

AAM-HEAP-INTERCLASABIL(G)

```
 1: T ← ∅
 2: pentru fiecare vârf v_i ∈ V[G] execută
 3:    V_i ← {v_i}
 4:    E_i ← {(v_i, v) ∈ E[G]}
 5: cât timp există mai mult decât o mulţime V_i execută
 6:    alege oricare din mulţimile V_i
 7:    extrage muchia (u, v) de pondere minimă din E_i
 8:    fără a pierde generalitatea presupunem că u ∈ V_i şi v ∈ V_j
 9:    dacă i ≠ j atunci
10:       T ← T ∪ {(u, v)}
11:       V_i ← V_i ∪ V_j, distrugând V_j
12:       E_i ← E_i ∪ E_j
```

Procedura precedentă, descrisă în pseudocod, construieşte un arbore de acoperire minim T. Demonstrarea corectitudinii ei se poate face cu metoda descrisă în secţiunea 24.1. Procedura construieşte o partiţie $\{V_i\}$ de vârfuri ale lui V şi pentru fiecare V_i, o mulţime

$$E_i \subseteq \{(u, v) : u \in V_i \text{ sau } v \in V_i\}$$

de muchii incidente vârfurilor din V_i.

Descrieţi cum se poate implementa acest algoritm folosind operaţiile pe heap-uri interclasabile prezentate în figura 20.1. Calculaţi timpul de execuţie pentru implementarea găsită, presupunând că operaţiile pe heap-uri interclasabile sunt implementate de heap-urile binomiale.

Note bibliografice

Heap-urile binomiale au fost introduse în 1978 de Vuillemin [196]. Brown [36, 37] a studiat în detaliu proprietăţile acestora.

21 Heap-uri Fibonacci

Am văzut în capitolul 20 că heap-urile binomiale suportă operaţiile INSEREAZĂ, MINIM, EXTRAGE-MIN şi REUNEŞTE, plus operaţiile DESCREŞTE-CHEIE şi ŞTERGE cu un timp de execuţie, în cazul cel mai defavorabil, de $O(\lg n)$. În acest capitol vom examina heap-urile Fibonacci care suportă aceleaşi operaţii şi au avantajul că operaţiile care nu implică ştergerea unui element, se execută într-un timp amortizat $O(1)$.

Heap-urile Fibonacci sunt preferabile, din punct de vedere teoretic, atunci când numărul operaţiilor EXTRAGE-MIN şi ŞTERGE este relativ mic în raport cu celelalte operaţii efectuate. Această situaţie se regăseşte în multe aplicaţii. De exemplu, mulţi algoritmi pentru probleme pe grafuri apelează DESCREŞTE-CHEIE o singură dată pentru fiecare muchie. Pentru grafuri dense, care au muchii multe, timpul amortizat $O(1)$, pentru fiecare apel DESCREŞTE-CHEIE înseamnă o îmbunătăţire considerabilă în raport cu $\Theta(\lg n)$, timpul, în cazul cel mai defavorabil, pentru implementările prin heap-uri binare sau binomiale. Pentru probleme ca şi calcularea arborilor de acoperire minimă (capitolul 24) şi determinarea drumurilor de lungimă minimă de sursă unică (capitolul 25), algoritmii actuali cei mai rapizi (asimptotic) folosesc heap-uri Fibonacci.

Totuşi, din punct de vedere practic, factorii constanţi şi complexitatea programării heap-urilor Fibonacci fac ca pentru multe aplicaţii să fie preferate heap-urile binare obişnuite (sau k-are) în locul celor Fibonacci. Astfel, heap-urile Fibonacci prezintă, în special, un interes teoretic. Descoperirea unei structuri de date mai simplă dar care să aibă margini amortizate de timp la fel ca şi heap-urile Fibonacci, ar fi de o importanţă practică considerabilă.

Un heap Fibonacci, la fel ca şi un heap binomial, este format dintr-o colecţie de arbori. Heap-urile Fibonacci se bazează de fapt pe heap-uri binomiale. Dacă pe un heap Fibonacci nu sunt invocate niciodată DESCREŞTE-CHEIE şi ŞTERGE, atunci fiecare arbore din heap este asemănător unui arbore binomial. Heap-urile Fibonacci diferă totuşi de heap-urile binomiale prin faptul că au o structură mai relaxată, permiţând astfel o îmbunătăţire asimptotică a marginilor de timp. Întreţinerea structurii poate fi amânată până când acest lucru se poate efectua convenabil.

Heap-urile Fibonacci se constituie a fi, la fel ca şi tabelele dinamice din secţiunea 18.4, un exemplu bun de structură de date proiectată având în vedere o analiză amortizată. Metoda de potenţial din secţiunea 18.3 va constitui în cele ce urmează suportul intuiţiei şi analizei operaţiilor pe heap-uri Fibonacci.

Expunerea din acest capitol presupune că aţi parcurs capitolul 20 despre heap-uri binomiale. Specificarea operaţiilor este dată în acel capitol, iar tabelul din figura 20.1 prezintă şi marginile de timp pentru operaţiile pe heap-uri binare, binomiale şi Fibonacci. În prezentarea structurii unui heap Fibonacci ne vom baza pe structura de heap binomial. Veţi observa că unele operaţii efectuate pe heap-uri Fibonacci sunt similare cu cele efectuate pe heap-uri binomiale.

La fel ca şi heap-urile binomiale, heap-urile Fibonacci nu sunt proiectate pentru a oferi un suport eficient operaţiei CAUTĂ; operaţiile care se referă la un anumit nod vor necesita astfel un pointer la acel nod.

Secţiunea 21.1 defineşte heap-urile Fibonacci, discută reprezentarea lor şi prezintă funcţia de potenţial care va fi folosită în analiza amortizată. Secţiunea 21.2 prezintă modul de implementare a operaţiilor pe heap-uri interclasabile, astfel încât să obţinem marginile de timp prezentate în figura 20.1. Cele două operaţii rămase, DESCREŞTE-CHEIE şi ŞTERGE, sunt prezentate în secţiunea 21.3. La final, secţiunea 21.4 încheie o parte esenţială a analizei.

21.1. Structura heap-urilor Fibonacci

Similar cu un heap binomial, un ***heap Fibonacci*** este o colecţie de arbori care au proprietatea de ordonare de heap. Totuşi, arborii dintr-un heap Fibonacci nu trebuie să fie arbori binomiali. Figura 21.1(a) prezintă un exemplu de heap Fibonacci.

Spre deosebire de arborii dintr-un heap binomial, care sunt ordonaţi, arborii aparţinând unui heap Fibonacci sunt arbori cu rădăcină, dar neordonaţi. După cum prezintă figura 21.1(b), fiecare nod x conţine un pointer $p[x]$ spre părintele lui şi un pointer $\mathit{fiu}[x]$ la unul din fiii lui. Fiii lui x sunt înlănţuiţi circular printr-o listă dublu înlănţuită, numită ***lista fiilor*** lui x. Fiecare fiu y dintr-o listă de fii are pointerii $\mathit{stânga}[y]$ şi $\mathit{dreapta}[y]$ care indică fratele stăng, respectiv drept al lui y. Dacă nodul y este singurul fiu, atunci $\mathit{stânga}[y] = \mathit{dreapta}[y] = y$. Ordinea în care apar fiii în lista de fii este arbitrară.

Folosirea listelor dublu înlănţuite şi circulare (a se vedea secţiunea 11.2) pentru heap-uri Fibonacci au două avantaje. În primul rând, putem şterge un nod dintr-o listă dublu înlănţuită şi circulară într-un timp $O(1)$. În al doilea rând, putem concatena două liste de acest tip, obţinând o singură listă dublu înlănţuită şi circulară tot într-un timp $O(1)$. În descrierea operaţiilor pe heap-uri Fibonacci ne vom referi informal la aceste operaţii pe liste, lăsând cititorul să completeze detaliile lor de implementare.

Fiecare nod va conţine încă două câmpuri utile. Numărul de fii din lista fiilor nodului x este memorat în $\mathit{grad}[x]$. Câmpul de tip boolean $\mathit{marcat}[x]$ va indica dacă nodul x a pierdut un fiu după ultima operaţie în care x a fost transformat în fiul unui alt nod. Până în secţiunea 21.3 nu ne vom îngriji de marcarea nodurilor. Nodurile nou create sunt nemarcate, iar un nod x devine nemarcat atunci când este transformat în fiul unui alt nod.

Un heap Fibonacci H este referit printr-un pointer $\mathit{min}[H]$ al rădăcinii arborelui care conţine cheia minimă; acest nod este numit ***nodul minim*** al heap-ului Fibonacci. Dacă un heap Fibonacci este vid atunci $\mathit{min}[H] = \text{NIL}$.

Rădăcinile tuturor arborilor dintr-un heap Fibonacci sunt înlănţuite prin pointerii $\mathit{stânga}$ şi $\mathit{dreapta}$, formând o listă dublu înlănţuită şi circulară, numită ***listă de rădăcini*** a heap-ului Fibonacci. Astfel, pointerul $\mathit{min}[H]$ indică acel nod din lista de rădăcini care are cheia minimă. Ordinea nodurilor în lista de rădăcini este arbitrară.

Un atribut pe care ne vom baza este şi numărul nodurilor $n[H]$ al unui heap Fibonacci H.

Funcţia de potenţial

După cum am menţionat, vom folosi în analiza complexităţii operaţiilor pe heap-uri Fibonacci metoda de potenţial din secţiunea 18.3. Pentru un heap Fibonacci H, vom indica prin $t(H)$ numărul rădăcinilor din lista de rădăcini a lui H, iar prin $m(H)$ numărul nodurilor marcate din H. Potenţialul unui heap Fibonacci este definit prin

$$\Phi(H) = t(H) + 2m(H). \tag{21.1}$$

De exemplu, potenţialul heap-ului Fibonacci din figura 21.1 este $5 + 2 \cdot 3 = 11$. Potenţialul unei mulţimi de heap-uri Fibonacci este egal cu suma potenţialelor heap-urilor Fibonacci conţinute. Vom presupune că o unitate de potenţial corespunde unui volum de calcul constant, cu o constantă suficient de mare care acoperă orice calcul necesar care se execută într-un timp constant.

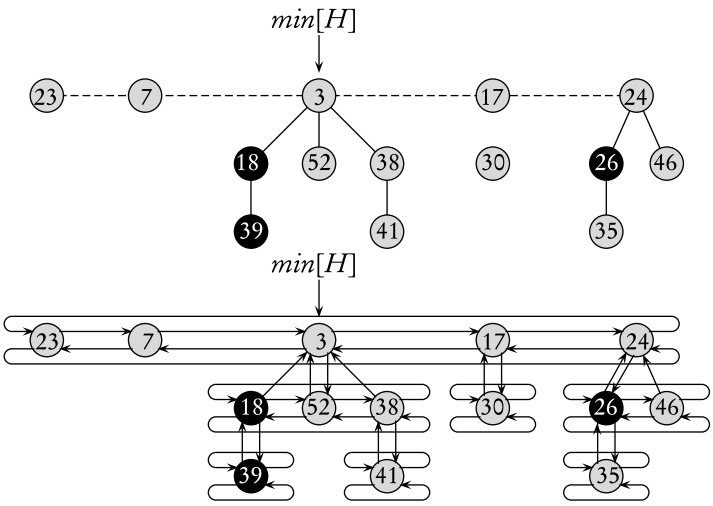


Figura 21.1 **(a)** Un heap Fibonacci constând din cinci arbori, ce satisfac proprietatea de heap ordonat, şi care este din 14 noduri. Linia punctată indică lista de rădăcini. Nodul care conţine valoarea 3 este nodul minim din heap. Nodurile marcate sunt haşurate cu negru. Potenţialul acestui heap Fibonacci este $5 + 2 \cdot 3 = 11$. **(b)** O reprezentare mai completă care arată pointerii p (săgeţile orientate în sus), *fiu* (săgeţile în jos), şi *stâng* şi *drept* (săgeţile laterale). Aceste detalii vor fi omise în celelelalte figuri ale capitolului, deoarece toate informaţiile prezentate aici pot fi determinate din ceea ce apare în partea (a).

Vom presupune că aplicaţiile pentru heap-uri Fibonacci nu pornesc de la nici un heap. Astfel, potenţialul iniţial este 0 şi conform ecuaţiei (21.1), potenţialul este nenegativ la orice moment de timp ulterior. Din ecuaţia (18.2), o margine superioară a costului amortizat total este şi o margine superioară a costului actual pentru secvenţa de operaţii.

Grad maxim

În secţiunile următoare ale capitolului vom efectua o analiză amortizată care se va baza pe cunoaşterea unei margini superioare $D(n)$ a gradului maxim al oricărui nod dintr-un arbore Fibonacci având n noduri. Exerciţiul 21.2-3 arată că $D(n) = \lfloor \lg n \rfloor$ atunci când avem doar operaţii de interclasare pe heap-uri. În secţiunea 21.3 vom arăta că, dacă includem şi operaţiile DESCREŞTE-CHEIE şi ŞTERGE, $D(n) = O(\lg n)$.

21.2. Operaţiile heap-urilor interclasabile

În această secţiune vom descrie şi analiza operaţiile heap-urilor interclasabile precum şi implementări pe heap-uri Fibonacci. Dacă se au în vedere doar aceste operaţii – CREEAZĂ-HEAP, INSEREAZĂ, MINIM, EXTRAGE-MIN şi REUNEŞTE – fiecare heap Fibonacci este o colecţie de arbori binomiali "neordonaţi". Un ***arbore binomial neordonat*** este asemănător unui arbore binomial şi se defineşte de asemenea recursiv. Arborele binomial neordonat U_0 constă dintr-un singur nod, iar un arbore binomial neordonat U_k constă din doi arbori binomiali neordonaţi U_{k-1},

rădăcina unuia din ei fiind (*oricare*) fiu al rădăcinii celuilalt. Proprietăţile arborilor binomiali – lema 20.1, sunt adevărate şi pentru arborii binomiali neordonaţi, înlocuind însă proprietatea 4 (a se vedea exerciţiul 21.2-2) cu:

4'. Gradul rădăcinii arborelui binomial neordonat U_k este k, grad mai mare decât al oricărui alt nod. Într-o ordine oarecare, descendenţii rădăcinii sunt rădăcinile unor subarbori U_0, U_1, ..., U_{k-1}.

Astfel, deoarece un heap Fibonacci având n noduri este o colecţie de arbori binomiali neordonaţi, $D(n) = \lg n$.

Ideea de bază, aplicată la definirea operaţiilor heap-urilor interclasabile pe heap-uri Fibonacci este de a întârzia prelucrările cât se poate de mult. Obţinerea unor operaţii performante poate fi în detrimentul altora. Dacă numărul arborilor unui heap Fibonacci este mic, atunci nodul minim necesar operaţiei EXTRAGE-MIN este determinat eficient. Dar, după cum s-a văzut şi pentru heap-uri binomiale – exerciţiul 20.2-10, costul asigurării unui număr mic de arbori este ridicat: pentru inserarea unui nod într-un heap binomial, sau pentru reuniunea a două heap-uri binomiale, timpul de execuţie se poate ridica până la $\Omega(\lg n)$. După cum se va vedea, la inserarea unui nod nou sau la reuniunea a două heap-uri nu vom încerca să consolidăm arborii din heap-ul Fibonacci. Consolidarea va fi lăsată în seama operaţiei EXTRAGE-MIN, operaţie care necesită găsirea nodului minim.

Crearea unui heap Fibonacci

Pentru a crea un heap Fibonacci vid, procedura CREEAZĂ-HEAP-FIB, creează şi returnează un obiect heap Fibonacci H, cu $n[H] = 0$ şi $\mathit{min}[H] = \text{NIL}$; H nu conţine arbori. Deoarece $t(H) = 0$ şi $m(H) = 0$, potenţialul heap-ului Fibonacci vid este $\Phi(H) = 0$. Costul amortizat al operaţiei CREEAZĂ-HEAP-FIB este astfel egal cu costul ei actual $O(1)$.

Inserarea unui nod

Procedura următoare inserează nodul x în heap-ul Fibonacci H, presupunând că nodul a fost alocat în prealabil şi că $\mathit{cheie}[x]$ a fost de asemenea iniţializată.

HEAP-FIB-INSEREAZĂ(H, x)
1: $\mathit{grad}[x] \leftarrow 0$
2: $\mathit{p}[x] \leftarrow \text{NIL}$
3: $\mathit{fiu}[x] \leftarrow \text{NIL}$
4: $\mathit{stânga}[x] \leftarrow x$
5: $\mathit{dreapta}[x] \leftarrow x$
6: $\mathit{marcat}[x] \leftarrow \text{FALS}$
7: concatenează lista de rădăcini care îl conţine pe x cu lista de rădăcini a lui H
8: **dacă** $\mathit{min}[H] = \text{NIL}$ sau $\mathit{cheie}[x] < \mathit{cheie}[\mathit{min}[H]]$ **atunci**
9: $\quad \mathit{min}[H] \leftarrow x$
10: $n[H] \leftarrow n[H] + 1$

După iniţializarea câmpurilor nodului x în liniile 1–6, făcându-l circular şi dublu înlănţuit, în linia 7 x este adăugat listei de rădăcini a lui H într-un timp actual $O(1)$. Astfel nodul x devine un arbore cu un singur nod şi care are proprietatea de ordonare de heap, deci un arbore

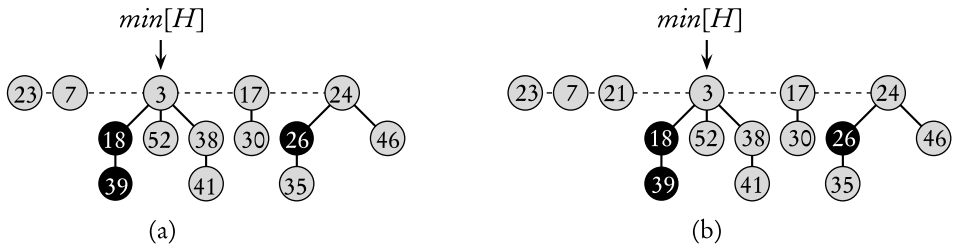


Figura 21.2 Inserarea unui nod într-un ansaamblu Fibonacci. **(a)** Un heap Fibonacci H **(b)** Heap-ul Fibonacci H după ce nodul având cheia 21 a fost inserat. Nodul devine un arbore cu proprietatea de heap şi este apoi adăugat listei de rădăcini, devenind fratele stâng al rădăcinii.

binomial neordonat aparţinând heap-ului Fibonacci. Acest arbore nu are fii şi nu este marcat. În continuare în liniile 8–9 este actualizat, dacă este necesar, pointerul la nodul minim din heap-ul Fibonacci H. În final, linia 10 incrementează $n[H]$ marcând inserarea unui nod nou. Figura 21.2 prezintă inserarea nodului având cheia 21 în heap-ul Fibonacci din figura 21.1.

Spre deosebire de HEAP-BINOMIAL-INSEREAZĂ, HEAP-FIB-INSEREAZĂ nu consolidează arborii din heap-ul Fibonacci. Dacă procedura HEAP-FIB-INSEREAZĂ este apelată consecutiv de k ori atunci se adaugă listei de rădăcini k arbori cu un singur nod.

Pentru a determina costul amortizat al operaţiei HEAP-FIB-INSEREAZĂ, fie H heap-ul Fibonacci dat ca intrare şi H' heap-ul Fibonacci rezultat. Atunci $t(H') = t(H) + 1$ şi $m(H') = m(H)$ şi creşterea potenţialului este

$$((t(H) + 1) + 2m(H)) - (t(h) + 2m(H)) = 1.$$

Deoarece costul actual este $O(1)$, costul amortizat este $O(1) + 1 = O(1)$.

Găsirea nodului minim

Nodul minim al unui heap Fibonacci H este dat de pointerul $min[H]$, de unde costul actual al operaţiei este $O(1)$. Deoarece potenţialul lui H nu se schimbă, costul amortizat al operaţiei este egal cu costul ei actual $O(1)$.

Reuniunea a două heap-uri Fibonacci

Procedura următoare reuneşte heap-urile Fibonacci H_1 şi H_2, distrugând H_1 şi H_2 în timpul desfăşurării procesului.

HEAP-FIB-REUNEŞTE(H_1, H_2)
1: $H \leftarrow$ CREEAZĂ-HEAP-FIB
2: $min[H] \leftarrow min[H_1]$
3: concatenează lista de rădăcini a lui H_2 cu lista de rădăcini a lui H
4: **dacă** ($min[H_1] =$ NIL) sau ($min[H_2] \neq$ NIL şi $min[H_2] < min[H_2]$) **atunci**
5: $\quad min[H] \leftarrow min[H_2]$
6: $n[H] \leftarrow n[H_1] + n[H_2]$
7: eliberează obiectele H_1 şi H_2
8: **returnează** H

Liniile 1–3 concatenează listele de rădăcini ale lui H_1 şi H_2 obţinându-se o listă de rădăcini nouă H. Liniile 2, 4 şi 5 stabilesc nodul minim al lui H, iar linia 6 iniţializează $n[H]$ cu numărul total de noduri. Obiectele H_1 şi H_2 sunt dealocate în linia 7, iar linia 8 returnează heap-ul Fibonacci H rezultat. Arborii nu sunt consolidaţi, la fel ca şi în procedura HEAP-FIB-INSEREAZĂ. Potenţialul rămâne neschimbat şi anume:

$$\begin{aligned}&\Phi(H) - (\Phi(H_1) + \Phi(H_2))\\ &= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) = 0,\end{aligned}$$

deoarece $t(H) = t(H_1) + t(H_2)$ şi $m(H) = m(H_1) + m(H_2)$. Costul amortizat al operaţiei HEAP-FIB-REUNEŞTE este astfel egal cu costul actual $O(1)$.

Extragerea nodului minim

Procesul de extragere al nodului minim este cel mai complicat dintre toate operaţiile prezentate în această secţiune. Consolidarea arborilor din lista de rădăcini este efectuată în cadrul acestei operaţii. Următorul algoritm scris în pseudocod extrage nodul minim. Algoritmul presupune că atunci când se şterge un nod dintr-o listă înlănţuită, pointerii rămaşi în listă sunt actualizaţi, dar pointerii din nodul eliminat sunt lăsaţi neschimbaţi. În algoritm se foloseşte o procedură auxiliară CONSOLIDEAZĂ care va fi prezentată pe scurt.

```
HEAP-FIB-EXTRAGE-MIN(H)
 1: z ← min[H]
 2: dacă z ≠ NIL atunci
 3:    pentru fiecare fiu x al lui z execută
 4:       adaugă x la lista de rădăcini a lui H
 5:       p[x] ← NIL
 6:    şterge z din lista de rădăcini a lui H
 7:    dacă z = dreapta[z] atunci
 8:       min[H] ← NIL
 9:    altfel
10:       min[H] ← dreapta[z]
11:       CONSOLIDEAZĂ(H)
12:    n[H] ← n[H] − 1
13: returnează z
```

Conform figurii 21.3, procedura HEAP-FIB-EXTRAGE-MIN rupe legăturile între nodul rădăcină minim respectiv fiii săi şi şterge nodul minim din lista de rădăcini. Apoi consolidează lista de rădăcini prin înlănţuirea rădăcinilor de acelaşi grad, până când rămâne cel mult o rădăcină de fiecare grad.

Pornim în linia 1 prin reţinerea unui pointer z la nodul minim; în final se va returna acest pointer. Dacă $z = \text{NIL}$ atunci heap-ul Fibonacci H este vid şi prelucrarea este încheiată. În caz contrar, la fel ca şi în procedura HEAP-BINOMIAL-EXTRAGE-MIN, nodul z este şters din H în liniile 3–5 prin transformarea fiilor lui z în rădăcini) şi ştergerea lui z din lista de rădăcini (linia 6). Dacă $z = \mathit{dreapta}[z]$ atunci z fusese singurul nod din lista de rădăcini şi nu avusese nici un fiu, astfel încât, mai întâi, trebuie să facem heap-ul H vid, înainte. În caz contrar, stabilim pointerul $\mathit{min}[H]$ în lista de rădăcini astfel încât să indice nodul minim rămas (în acest caz,

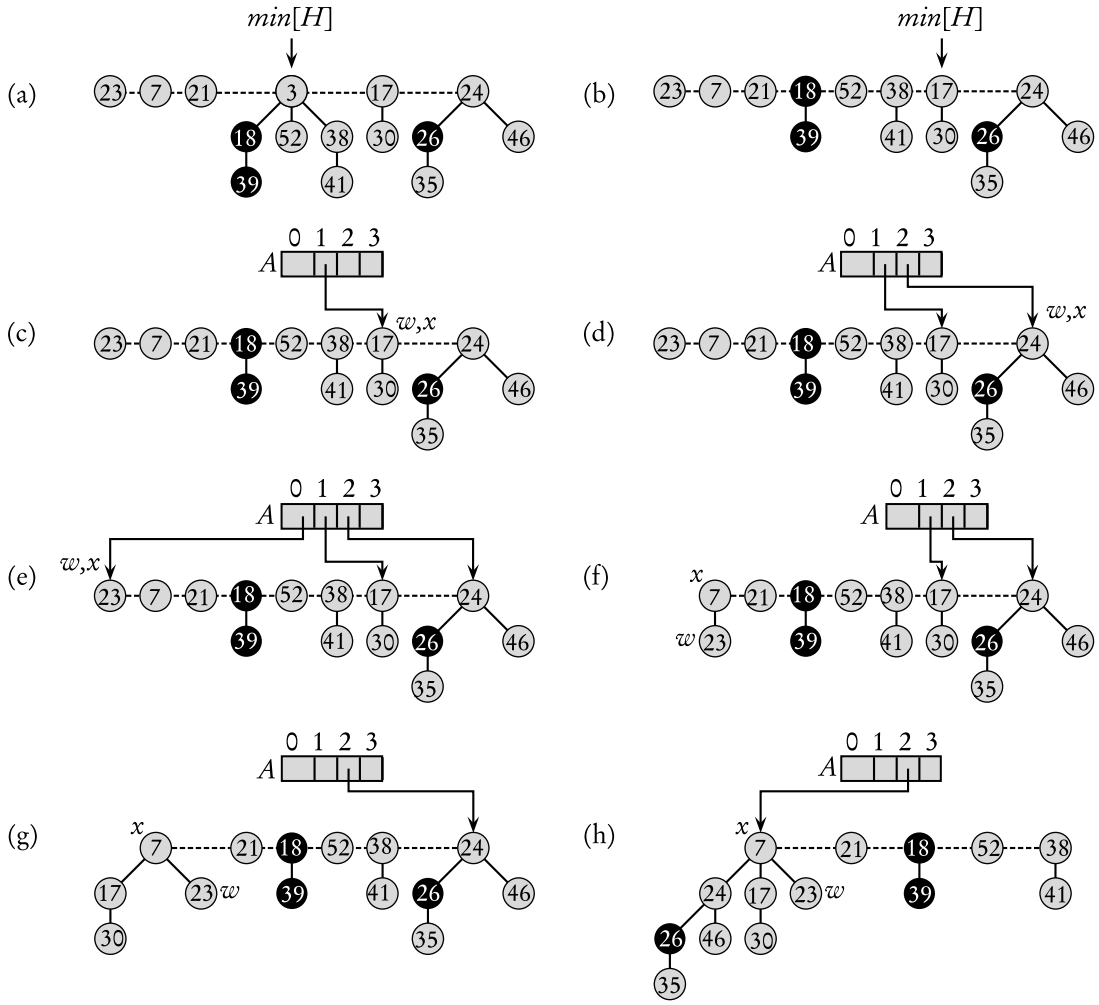
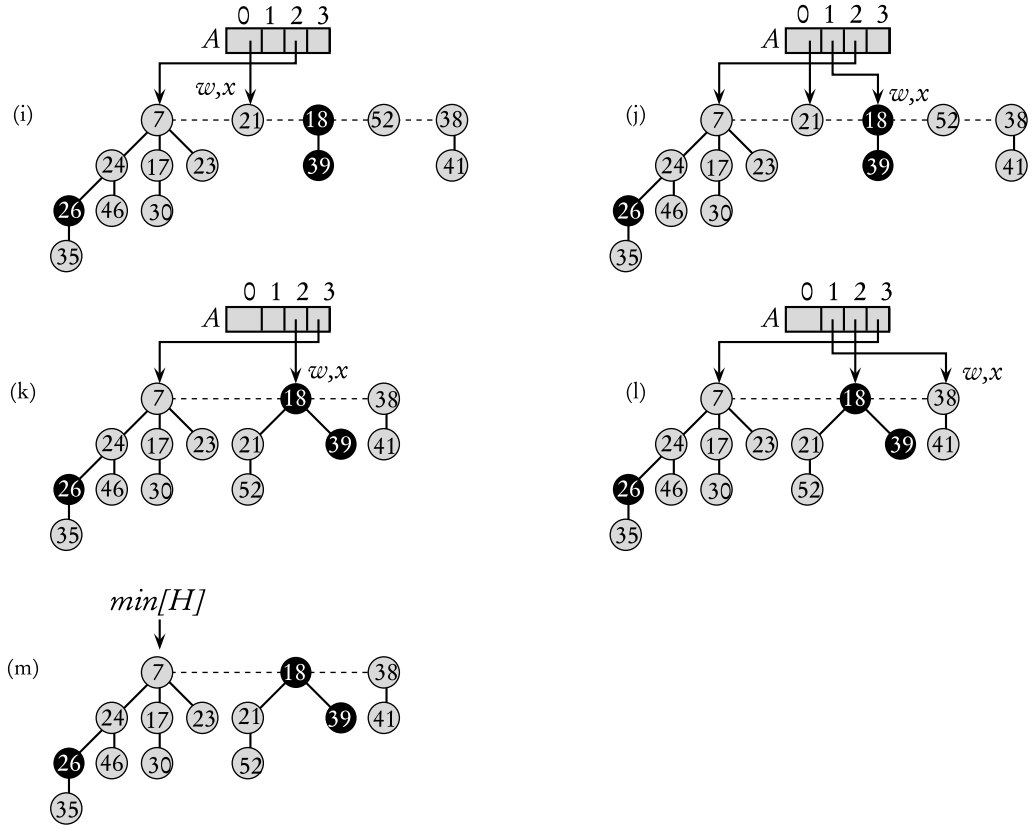


Figura 21.3 Acţiunile procedurii HEAP-FIB-EXTRAGE-MIN. **(a)** Un heap Fibonacci H. **(b)** Situaţia obţinută după ştergerea nodului minim z din lista de rădăcini şi adăugarea la lista de rădăcini a fiilor lui z. **(c)–(e)** Tabloul A şi arborii după primele trei iteraţii ale ciclului **pentru** din liniile 3–13 ale procedurii CONSOLIDEAZĂ. Lista de rădăcini este procesată pornind de la nodul minim şi urmând pointerii *dreapta*. Fiecare parte arată valorile lui w şi x la sfârşitul unei iteraţii. **(f)–(h)** Iteraţia următoare a ciclului **pentru**, cu valorile lui w şi x obţinute la sfârşitul fiecărei iteraţii a ciclului **cât timp** din liniile 6–12. Partea (f) arată situaţia obţinută la prima trecere prin ciclul **cât timp**. Nodul cu cheia 23 a fost legat la nodul având cheia 7, nod care este indicat acum de x. În partea (g) nodul având cheia 17 a fost legat la nodul având cheia 7, spre care încă indică x. În partea (h) nodul având cheia 24 a fost legat la nodul având cheia 7. Deoarece $A[3]$ nu indică nici un nod, la sfârşitul iteraţiei ciclului **pentru** $A[3]$ va indica rădăcina arborelui rezultat. **(i)–(l)** Situaţia obţinută după următoarele patru iteraţii ale ciclului **cât timp**. **(m)** Heap-ul Fibonacci H după reconstruirea listei de rădăcini din tabloul A şi după determinarea pointerului $min[H]$.



dreapta[z]) diferit de z. Figura 21.3(b) prezintă heap-ul Fibonacci din figura 21.3(a) după executarea instrucţiunii din linia 9.

Pasul următor în care reducem numărul de arbori din heap-ul Fibonacci constă în ***consolidarea*** listei de rădăcini a lui H; acesta este efectuat prin apelul CONSOLIDEAZĂ(H). Consolidarea listei de rădăcini constă din efectuarea repetată a paşilor următori, până când fiecare rădăcină din lista de rădăcini are o valoare distinctă pentru *gradul* său.

1. Găseşte două rădăcini x şi y din lista de rădăcini având acelaşi grad şi *cheie*[x] $\leq$ *cheie*[y].
2. ***Înlănţuie*** y la x: şterge y din lista de rădăcini şi include nodul y printre fiii lui x. Această operaţie este efecuată prin procedura HEAP-FIB-ÎNLĂNŢUIE. Câmpul *grad*[x] este incrementat, iar marcajul nodului y, dacă există, este şters.

Procedura CONSOLIDEAZĂ foloseşte un tablou auxiliar $A[0..D(n[H])]$; dacă $A[i] = y$ atunci y este o rădăcină cu $grad[y] = i$. Aceasta funcţionează după cum urmează. În liniile 1–2 se iniţializează A atribuind fiecărui element valoarea NIL. Procesarea fiecărui nod w se încheie cu un nod x care poate fi, sau nu, identic cu w. Astfel, elementele tabloului A[*grad*[x]] sunt iniţializate cu x. În ciclul **pentru** din liniile 3–13 este examinat fiecare nod w din lista de rădăcini. Invariantul la fiecare iteraţie în ciclul **for** este că nodul x este rădăcina acelui arbore care conţine nodul w.

CONSOLIDEAZĂ(H)

```
 1: pentru i ← 0, D(n[H]) execută
 2:    A[i] ← NIL
 3: pentru fiecare nod w din lista de rădăcini a lui H execută
 4:    x ← w
 5:    d ← grad[x]
 6:    cât timp A[d] ≠ NIL execută
 7:       y ← A[d]
 8:       dacă cheie[x] > cheie[y] atunci
 9:          interschimbă x ↔ y
10:       HEAP-FIB-ÎNLĂNŢUIE(H, y, x)
11:       A[d] ← NIL
12:       d ← d + 1
13:    A[d] ← x
14: min[H] ← NIL
15: pentru i ← 0, D(n[H]) execută
16:    dacă A[i] ≠ NIL atunci
17:       adaugă A[i] listei de rădăcini a lui H
18:       dacă min[H] = NIL sau cheie[A[i]] < cheie[min[H]] atunci
19:          min[H] ← A[i]
```

HEAP-FIB-ÎNLĂNŢUIE(H, y, x)

```
1: şterge y din lista de rădăcini a lui H
2: înlănţuie y ca fiu al lui x şi incrementează grad[x]
3: marcat[x] ← FALS
```

Ciclul **cât timp** din liniile 6–12 are predicatul invariant $d = \mathit{grad}[x]$ (cu excepţia liniei 11 după cum vom vedea imediat). La fiecare iteraţie a ciclului **cât timp** $A[d]$ indică o anumită rădăcină y. Deoarece $d = \mathit{grad}[x] = \mathit{grad}[y]$, vom înlănţui x şi y. Cel care are cheia mai mică dintre x şi y devine părintele celuilalt, în urma operaţiei de înlănţuire; astfel dacă este necesar, pointerii x şi y sunt interschimbaţi în liniile 8–9. În continuare y este legat la x prin apelul din linia 10, HEAP-FIB-ÎNLĂNŢUIE(H, y, x). În urma apelului, $\mathit{grad}[x]$ este incrementat, iar $\mathit{grad}[y]$ rămâne d. Deoarece nodul y nu mai este rădăcină, pointerul spre el din tabloul A este şters în linia 11. Deoarece după apelul HEAP-FIB-ÎNLĂNŢUIE valoarea $\mathit{grad}[x]$ este incrementată, în linia 12 este restabilită proprietatea invariantului $d = \mathit{grad}[x]$. Ciclul **cât timp** este executat repetat până când $A[d] =$ NIL, situaţie în care nu există alte rădăcini având acelaşi grad ca şi x. În linia 13 iniţializăm $A[d]$ cu x şi efectuăm iteraţia următoare a ciclului **pentru**. Figurile 21.3(c)–(e) prezintă tabloul A şi arborii rezultaţi după primele trei iteraţii ale ciclului **pentru** din liniile 3–13. La iteraţia următoare a ciclului **pentru** sunt realizate trei legături; rezultatele lor se pot vedea în figurile 21.3(f)–(h). Figurile 21.3(i)–(l) prezintă rezultatele următoarelor patru iteraţii ale ciclului **pentru**.

Rămâne să definitivăm operaţiile începute. După execuţia ciclului **pentru** din liniile 3–13, linia 14 videază lista de rădăcini iar în liniile 15–19 aceasta este reconstruită. Heap-ul Fibonacci rezultat este redat în figura 21.3(m). După consolidarea listei de rădăcini, operaţiile efectuate de HEAP-FIB-EXTRAGE-MIN se încheie cu decrementarea valorii $n[H]$ în linia 11 şi returnarea în linia 12 a unui pointer la nodul şters z.

Observăm că dacă anterior apelului HEAP-FIB-EXTRAGE-MIN toţi arborii din heap-ul

Fibonacci sunt arbori binomiali neordonaţi, atunci arborii rezultaţi după apel sunt de asemenea binomiali neordonaţi. Arborii sunt modificaţi în două feluri. În primul rând, în liniile 3–5 din HEAP-FIB-EXTRAGE-MIN, fiecare fiu x al rădăcinii z devine o rădăcină. Conform exerciţiului 21.2-2 fiecare arbore nou va fi un arbore binomial neordonat. În al doilea rând, arborii sunt înlănţuiţi prin HEAP-FIB-ÎNLĂNŢUIE doar dacă sunt de acelaşi grad. Deoarece înainte de crearea legăturilor toţi arborii sunt binomiali şi neordonaţi, cei doi arbori cu k fii au fiecare o structură de tip U_k. Rezultă că arborele obţinut va avea o structură U_{k+1}.

Acum vom arăta că extragerea nodului minim dintr-un heap Fibonacci având n noduri are un cost amortizat $O(D(n))$. Fie H heap-ul Fibonacci pentru care aplicăm operaţia HEAP-FIB-EXTRAGE-MIN.

Calcularea costului actual al extragerii nodului minim poate fi făcută după cum urmează. O contribuţie $O(D(n))$ provine din faptul că nodul minim are cel mult $D(n)$ fii care sunt procesaţi de HEAP-FIB-EXTRAGE-MIN şi din calculele efectuate în liniile 1–2 şi 14–19 în procedura CONSOLIDEAZĂ. Ne rămâne să analizăm contribuţia din partea ciclului **pentru** din liniile 3–13. După apelul procedurii CONSOLIDEAZĂ lungimea listei de rădăcini este cel mult $D(n)+t(H)-1$, deoarece constă din nodurile iniţiale ale listei în număr de $t(H)$, mai puţin nodul rădăcină extras şi plus numărul fiilor nodului extras care pot fi cel mult $D(n)$. De fiecare dată, în ciclul **cât timp** din liniile 6–12 una din rădăcini este înlănţuită cu alta, astfel calculul total efectuat de ciclul **pentru** este cel mult proporţional cu $D(n) + t(H)$. Astfel, timpul total de execuţie este $O(D(n) + t(H))$.

Înaintea extragerii nodului minim potenţialul este $t(H) + 2m(H)$, iar după extragere acesta este cel mult egal cu $(D(n) + 1) + 2m(H)$, deoarece rămân cel mult $D(n) + 1$ rădăcini şi nu se marchează noduri în timpul efectuării operaţiilor. Astfel costul amortizat este cel mult

$$\begin{aligned} &O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\ &= O(D(n)) + O(t(H)) - t(H) = O(D(n)), \end{aligned}$$

deoarece putem scala unităţile de potenţial pentru a depăşi constanta ascunsă în $O(t(H))$. Intuitiv, costul efectuării fiecărei legături se datorează reducerii potenţialului în urma înlănţuirilor care micşorează cu unu numărul de rădăcini.

Exerciţii

21.2-1 Determinaţi heap-ul Fibonacci rezultat după apelul HEAP-FIB-EXTRAGE-MIN asupra heap-ului Fibonacci din figura 21.3(m).

21.2-2 Arătaţi că lema 20.1 are loc şi pentru arbori binomiali neordonaţi înlocuind însă proprietatea 4 cu 4'.

21.2-3 Arătaţi că dacă sunt suportate doar operaţiile pe heap-uri interclasabile, atunci gradul maxim $D(n)$ într-un heap Fibonacci având n noduri este cel mult $\lfloor \lg n \rfloor$.

21.2-4 Profesorul McGee a descoperit o nouă structură de date bazată pe heap-uri Fibonacci. Un heap McGee are aceeaşi structură ca şi un heap Fibonacci şi suportă operaţiile pe heap-uri interclasabile. Implementările operaţiilor sunt aceleaşi, cu menţiunea că inserarea şi reuniunea efectuează ca ultim pas operaţia de consolidare. Care sunt timpii de execuţie, în cazul cel mai defavorabil, ai operaţiilor pe heap-uri McGee? Este nouă structura descoperită de profesor?

21.2-5 Argumentaţi faptul că dacă singurele operaţii asupra cheilor sunt cele de comparare a două chei (ca în acest capitol), atunci nu toate operaţiile pe heap-uri interclasabile pot fi efectuate într-un timp amortizat $O(1)$.

21.3. Descreşterea unei chei şi ştergerea unui nod

În această secţiune vom arăta cum putem descreşte cheia unui nod dintr-un heap Fibonacci într-un timp amortizat $O(1)$ şi cum putem şterge oricare dintre nodurile unui heap Fibonacci având n noduri într-un timp amortizat $O(D(n))$. Aceste operaţii nu păstrează proprietatea că toţi arborii din heap sunt arbori binomiali neordonaţi. Putem mărgini gradul maxim $D(n)$ prin $O(\lg n)$, aceste valori fiind suficient de apropiate. Demonstrarea acestei margini va implica faptul că procedura HEAP-FIB-EXTRAGE-MIN şi HEAP-FIB-ŞTERGE se execută într-un timp amortizat $O(\lg n)$.

Descreşterea unei chei

În algoritmul pseudocod descris în continuare pentru HEAP-FIB-DESCREŞTE-CHEIE vom presupune din nou că ştergerea unui nod dintr-o listă înlănţuită nu modifică câmpurile nodului şters.

HEAP-FIB-DESCREŞTE-CHEIE(H, x, k)
1: **dacă** $k >$ *cheie*$[x]$ **atunci**
2: **eroare** "cheia nouă este mai mare decât cheia curentă"
3: *cheie*$[x] \leftarrow k$
4: $y \leftarrow p[x]$
5: **dacă** $y \neq$ NIL şi *cheie*$[x] <$ *cheie*$[y]$ **atunci**
6: TAIE(H, x, y)
7: TAIE-ÎN-CASCADĂ(H, y)
8: **dacă** *cheie*$[x] <$ *cheie*$[$*min*$[H]]$ **atunci**
9: *min*$[H] \leftarrow x$

TAIE(H, x, y)
1: şterge x din lista de fii ai lui y şi decrementează *grad*$[y]$
2: adaugă x la lista de rădăcini a lui H
3: $p[x] \leftarrow$ NIL
4: *marcat*$[x] \leftarrow$ FALS

TAIE-ÎN-CASCADĂ(H, y)
1: $z \leftarrow p[y]$
2: **dacă** $z \neq$ NIL **atunci**
3: **dacă** *marcat*$[y] =$ FALS **atunci**
4: *marcat*$[y] \leftarrow$ ADEVĂRAT
5: **altfel**
6: TAIE(H, y, z)
7: TAIE-ÎN-CASCADĂ(H, z)

HEAP-FIB-DESCREŞTE-CHEIE procedează după cum urmează. Liniile 1–3 asigură faptul că noua cheie nu este mai mare decât cheia curentă a lui x şi apoi atribuie această valoare lui x. Dacă x este o rădăcină sau dacă $\mathit{cheie}[x] \geq \mathit{cheie}[y]$, unde y este părintele lui x, atunci nu sunt necesare modificări structurale, deoarece proprietatea de heap ordonat nu este încălcată. Această condiţie este testată în liniile 4–5.

Dacă este încălcată proprietatea de heap ordonat, se pot petrece multe schimbări. Începem prin a ***tăia*** nodul x în linia 6. Procedura TAIE "taie" legătura dintre x şi părintele y al lui, transformându-l pe x în rădăcină.

Folosim câmpul *marcat* pentru a obţine marginile de timp dorite. Acestea ne ajută să obţinem următorul efect. Presupunem că x este un nod caracterizat de următoarele:

1. x fusese rădăcină la un anumit moment,
2. apoi x a fost legat la un alt nod,
3. apoi doi fii ai lui x au fost şterşi prin tăiere.

Imediat după ce şi-a pierdut şi al doilea fiu, lui x i se taie legătura cu părintele şi devine o nouă rădăcină. Câmpul $\mathit{marcat}[x]$ este ADEVĂRAT dacă au avut loc paşii 1 şi 2 şi un fiu al lui x a fost eliminat. Astfel, îndeplinind pasul 1, procedura TAIE şterge $\mathit{marcat}[x]$ în linia 4. (Putem observa acum de ce linia 3 din HEAP-FIB-ÎNLĂNŢUIE şterge $\mathit{marcat}[y]$: nodul y este legat la un alt nod şi astfel este efectuat pasul 2. Data următoare când se taie un fiu al lui y, lui $\mathit{marcat}[y]$ i se atribuie valoarea ADEVĂRAT.)

Încă nu am terminat, deoarece x ar putea fi al doilea fiu înlăturat al părintelui lui y, din momentul în care y a fost legat de un alt nod. Aşadar linia 7 din HEAP-FIB-DESCREŞTE-CHEIE efectuează o operaţie de ***tăiere în cascadă*** relativ la y. Dacă y este o rădăcină, atunci testul din linia 2 a operaţiei TAIE-ÎN-CASCADĂ are ca efect ieşirea din procedură. Dacă y nu este marcat, atunci procedura îl marchează în linia 4, deoarece primul lui fiu a fost tocmai tăiat, şi apelul se încheie. Totuşi, dacă y este marcat, înseamnă că tocmai şi-a pierdut al doilea fiu; în linia 6, y este înlăturat şi TAIE-ÎN-CASCADĂ se apelează recursiv în linia 7 asupra părintelui z al lui y. Apelul recursiv al procedurii TAIE-ÎN-CASCADĂ se desfăşoară ascendeent în arbore până se întâlneşte fie o rădăcină, fie un nod nemarcat.

O dată încheiat procesul de tăiere în cascadă, liniile 8–9 încheie HEAP-FIB-DESCREŞTE-CHEIE prin actualizarea lui $\mathit{min}[H]$, dacă este necesar.

Figura 21.4 prezintă rezultatul a două apeluri HEAP-FIB-DESCREŞTE-CHEIE aplicate pentru heap-ul Fibonacci din figura 21.4(a). Primul apel, redat în figura 21.4(b) nu implică tăieri în cascadă. Al doilea apel implică două tăieri în cascadă şi este redat în figurile 21.4(c)–(e).

Vom arăta acum că timpul amortizat pentru HEAP-FIB-DESCREŞTE-CHEIE este doar $O(1)$. Începem prin determinarea costului actual. Procedura HEAP-FIB-DESCREŞTE-CHEIE necesită un timp $O(1)$ plus timpul necesar efectuării tăierilor în cascadă. Presupunem că procedura TAIE-ÎN-CASCADĂ este apelată recursiv de c ori de la o anumită invocare a procedurii HEAP-FIB-DESCREŞTE-CHEIE. Fiecare apel TAIE-ÎN-CASCADĂ necesită un timp $O(1)$ fără a lua în considerare apelurile recursive. Astfel, timpul actual pentru HEAP-FIB-DESCREŞTE-CHEIE este $O(c)$ incluzând aici toate apelurile recursive.

În continuare vom calcula modificarea potenţialului. Fie H heap-ul Fibonacci înainte de apelul procedurii HEAP-FIB-DESCREŞTE-CHEIE. Fiecare apel recursiv al procedurii TAIE-ÎN-CASCADĂ, exceptându-l pe ultimul, înlătură un nod marcat şi şterge marcajul nodului. După

aceea, există $t(H) + c$ arbori ($t(H)$ arbori iniţiali, $c - 1$ arbori produşi de tăierea în cascadă şi arborele cu rădăcina x) şi cel mult $m(H) - c + 2$ noduri marcate ($c - 1$ au fost demarcate prin tăierea în cascadă, iar la ultimul apel, TAIE-ÎN-CASCADĂ marchează un nod). De unde, potenţialul se modifică cu cel mult

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c.$$

Astfel, costul amortizat pentru HEAP-FIB-DESCREŞTE-CHEIE este cel mult

$$O(c) + 4 - c = O(1),$$

deoarece putem scala unitatea de potenţial astfel încât să dominăm constanta ascunsă în $O(c)$.

Putem observa acum de ce funcţia potenţial a fost definită să conţină un termen care este de două ori mai mare decât nodurile marcate. Atunci când un nod marcat y este înlăturat la tăierea în cascadă, marcajul lui este şters, astfel că potenţialul se reduce cu 2. O unitate de potenţial este cheltuită pentru tăiere şi ştergerea marcajului, iar cealaltă unitate compensează creşterea în unităţi de potenţial datorată faptului că nodul y a devenit o rădăcină.

Ştergerea unui nod

A şterge un nod dintr-un heap Fibonacci având n noduri într-un timp amortizat $O(D(n))$ este uşor, după cum arată algoritmul pseudocod următor. Presupunem că heap-ul Fibonacci nu conţine chei de valoare $-\infty$.

HEAP-FIB-ŞTERGE(H, x)
1: HEAP-FIB-DESCREŞTE-CHEIE($H, x, -\infty$)
2: HEAP-FIB-EXTRAGE-MIN(H)

Procedura HEAP-FIB-ŞTERGE este analogă cu HEAP-BINOMIAL-ŞTERGE. Atribuind cheii nodului x valoarea $-\infty$, nodul x devine nodul minim din heap. Nodul x este şters apoi din heap prin apelul procedurii HEAP-FIB-EXTRAGE-MIN. Timpul amortizat pentru HEAP-FIB-ŞTERGE este suma dintre timpul amortizat $O(1)$ al procedurii HEAP-FIB-DESCREŞTE-CHEIE, respectiv $O(D(n))$ al lui HEAP-FIB-EXTRAGE-MIN.

Exerciţii

21.3-1 Presupunem că într-un heap Fibonacci o rădăcină x este marcată. Explicaţi cum a devenit x o rădăcină marcată. Justificaţi că pentru analiză nu are importanţă acest fapt, chiar dacă nu este o rădăcină care a fost întâi legată la un alt nod şi apoi şi-a pierdut fiul stâng.

21.3-2 Demonstraţi timpul amortizat $O(1)$ pentru HEAP-FIB-DESCREŞTE-CHEIE, folosind metoda agregării din secţiunea 18.1.

21.4. Mărginirea gradului maxim

Pentru a demonstra că timpul amortizat al operaţiilor HEAP-FIB-EXTRAGE-MIN şi HEAP-FIB-ŞTERGE este $O(\lg n)$, trebuie să arătăm că marginea superioară $D(n)$ a gradului oricărui

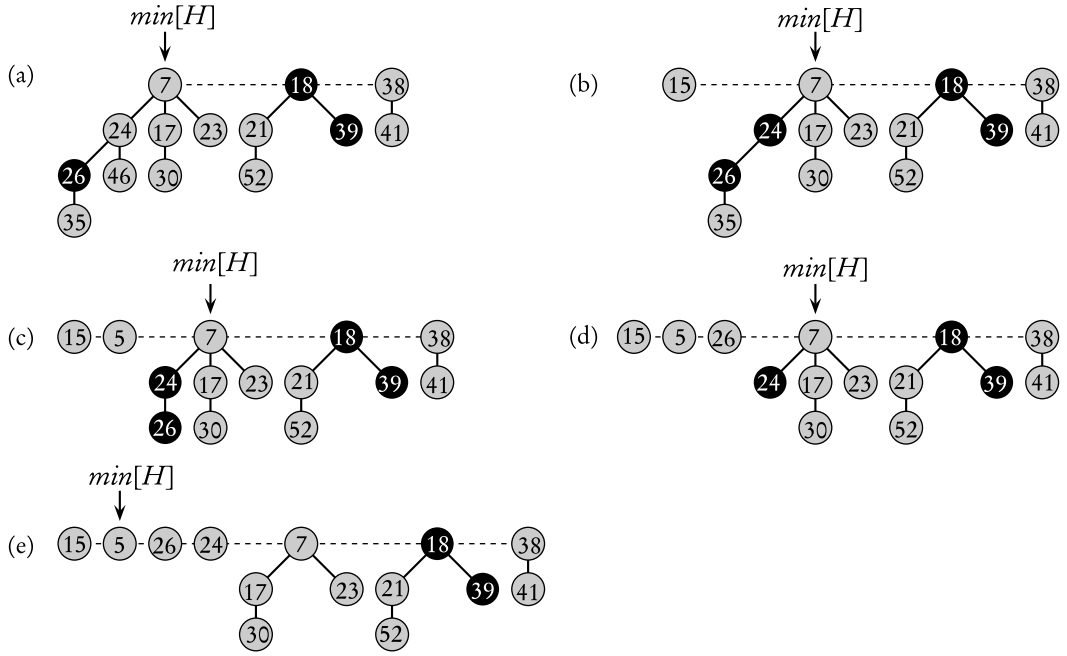


Figura 21.4 Două apeluri ale procedurii HEAP-FIB-DESCREŞTE-CHEIE. **(a)** Heap-ul Fibonacci iniţial. **(b)** Nodului având cheia 46 i se micşorează cheia la 15. Nodul devine o rădăcină şi părintele lui (având cheia 24) se marchează (anterior fusese nemarcat). **(c)**–**(e)** Nodului având cheia 35 i se micşorează cheia la 5. În partea (c), nodul (acum având cheia 5) devine o rădăcină. Părintele lui (având cheia 26) este marcat, urmând astfel o tăiere în cascadă. Nodul având cheia 26 este înlăturat de la părintele lui şi devine o rădăcină nemarcată (partea (d)). Are loc încă o tăiere în cascadă, deoarece nodul având cheia 24 este de asemenea marcat. Acest nod este înlăturat de la părintele lui şi devine o rădăcină nemarcatăa (partea (e)). Tăierile în cascadă se opresc aici deoarece nodul având cheia 7 este o rădăcină. (Chiar dacă nodul nu ar fi o rădăcină, procesul de tăiere s-ar fi încheiat, deoarece nodul este nemarcat.) Rezultatul operaţiei HEAP-FIB-DESCREŞTE-CHEIE este redat în partea (e), având $min[H]$ indicând noul nod minim.

nod dintr-un heap Fibonacci având n noduri este $O(\lg n)$. Conform exerciţiului 21.2-3, atunci când toţi arborii din heap-ul Fibonacci sunt arbori binomiali neordonaţi, $D(n) = \lfloor \lg n \rfloor$. Totuşi, tăierile care se efectuează în HEAP-FIB-DESCREŞTE-CHEIE pot genera arbori care nu respectă proprietăţile arborilor binomiali neordonaţi. În această secţiune vom arăta că prin înlăturarea unui nod de la părintele lui, dacă acesta pierde doi fii, atunci $D(n)$ este $O(\lg n)$. Mai precis, vom arăta că $D(n) \leq \lfloor \log_\phi n \rfloor$, unde $\phi = (1 + \sqrt{5})/2$.

Ideea de bază a analizei este următoarea. Pentru fiecare nod x dintr-un heap Fibonacci fie $\dim(x)$ numărul nodurilor subarborelui cu rădăcina x, incluzându-l şi pe x. (Să observăm că x nu trebuie să fie în lista de rădăcini – poate fi orice nod.) Vom arăta că $\dim(x)$ depinde exponenţial de $grad[x]$. Să reţinem că $grad[x]$ este actualizat pentru a exprima cu precizie gradul lui x.

Lema 21.1 Fie x un nod oarecare dintr-un heap Fibonacci şi presupunem că $grad[x] = k$. Notăm cu $y_1, y_2, \ldots, y_k$ fiii lui x în ordinea în care au fost legaţi la x, de la cel mai devreme spre cel mai recent. Atunci, $grad[y_1] \geq 0$ şi $grad[y_i] \geq i - 2$ pentru $i = 2, 3, \ldots, k$.

Demonstraţie. Este evident că $\mathit{grad}[y_1] \geq 0$.

Pentru $i \geq 2$, să observăm că atunci când y_i a fost legat la x, celelalte noduri $y_1, y_2, \ldots, y_{i-1}$ erau deja fii ai lui x, astfel că $\mathit{grad}[x] \geq i-1$. Nodul y_i este legat la x numai dacă $\mathit{grad}[x] = \mathit{grad}[y_i]$, de unde rezultă că la acel moment are loc de asemenea $\mathit{grad}[y_i] \geq i-1$. Din acel moment nodul y_i poate pierde cel mult un fiu, deoarece, dacă şi-ar fi pierdut ambii fii, ar fi fost tăiată legătura sa cu x. În concluzie, $\mathit{grad}[y_i] \geq i-2$. ∎

Ajungem în sfârşit la acea parte a analizei care justifică numele "heap-uri Fibonacci". Să ne amintim din secţiunea 2.2 că pentru $k = 0, 1, 2, \ldots$, al k-lea număr Fibonacci este definit prin relaţia de recurenţă

$$F_k = \begin{cases} 0 & \text{dacă } k = 0, \\ 1 & \text{dacă } k = 1, \\ F_{k-1} + F_{k-2} & \text{dacă } k \geq 2. \end{cases}$$

Lema următoare oferă un alt mod de a exprima F_k.

Lema 21.2 Pentru orice $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i.$$

Demonstraţie. Demonstraţia este prin inducţie după k. Dacă $k = 0$,

$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = 1 = F_2.$$

Presupunând acum că are loc ipoteza inducţiei, $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, avem

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k} F_i\right) = 1 + \sum_{i=0}^{k} F_i.$$

∎

Lema următoare şi corolarul ei completează analiza. Lema foloseşte inegalitatea (demonstrată în exerciţiul 2.2-8)

$$F_{k+2} \geq \phi^k,$$

unde ϕ este raportul de aur definit prin ecuaţia (2.14) ca fiind $\phi = (1 + \sqrt{5})/2 = 1.61803\ldots$.

Lema 21.3 Fie x un nod oarecare dintr-un heap Fibonacci şi fie $k = \mathrm{grad}[x]$. Atunci, $\dim(x) \geq F_{k+2} \geq \phi^k$, unde $\phi = (1 + \sqrt{5})/2$.

Demonstraţie. Notăm cu s_k valoarea minimă posibilă pentru $\dim(z)$ pentru toate nodurile z care au proprietatea $\mathit{grad}[z] = k$. Evident că, $s_0 = 1$, $s_1 = 2$ şi $s_2 = 3$. Numărul s_k este cel mult $\dim(x)$. Ca şi în lema 21.1, notăm cu $y_1, y_2, \ldots, y_k$ fiii lui x în ordinea în care au fost legaţi la x. Pentru a calcula o margine inferioară pentru $\dim(x)$ adunăm unu pentru x şi unu pentru primul fiu y_1 (pentru care $\dim(y_1) \geq 1$) şi aplicăm apoi lema 21.1 pentru ceilalţi fii. Avem astfel

$$\dim(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}.$$

Arătăm acum, prin inducţie în raport cu k, că $s_k \geq F_{k+2}$, pentru orice k întreg nenegativ. Cazurile de bază pentru $k = 0$ şi $k = 1$ sunt evidente. Pentru pasul inductiv, presupunem că $k \geq 2$ şi că $s_i \geq F_{i+2}$, pentru $i = 0, 1, \ldots, k-1$. Avem

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}.$$

Ultima egalitate rezultă din lema 21.2.
Am arătat astfel că $\dim(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ∎

Corolarul 21.4 Gradul maxim $D(n)$ al oricărui nod dintr-un heap Fibonacci cu n noduri, este $O(\lg n)$.

Demonstraţie. Fie x un nod oarecare al unui heap Fibonacci cu n noduri şi fie $k = \mathit{grad}[x]$. Din lema 21.3 obţinem că $n \geq \dim(x) \geq \phi^k$. Logaritmând în baza ϕ obţinem $k \leq \log_\phi n$. (De fapt $k \leq \lfloor \log_\phi n \rfloor$ deoarece k este întreg.) Gradul maxim $D(n)$ al oricărui nod este astfel $O(\lg n)$. ∎

Exerciţii

21.4-1 Profesorul Pinochio afirmă că înălţimea unui heap Fibonacci având n noduri este $O(\lg n)$. Demonstraţi că presupunerea profesorului este greşită descriind pentru orice număr întreg pozitiv n o secvenţă de operaţii ale heap-urilor Fibonacci care creează un heap Fibonacci constând dintr-un singur arbore care are cele n noduri înlănţuite liniar.

21.4-2 Presupunem că generalizăm regula de tăiere în cascadă a unui nod de la părintele lui, imediat ce acesta îşi pierde al k-lea fiu, pentru o constantă întreagă k fixată. (Regula din secţiunea 21.3 foloseşte $k = 2$.) Pentru ce valori ale lui k avem $D(n) = O(\lg n)$?

Probleme

21-1 Implementare alternativă pentru ştergere
Profesorul Pisano a propus următoarea variantă pentru procedura HEAP-FIB-ŞTERGE, afirmând că execuţia este mai rapidă atunci când nodul care se şterge nu este nodul indicat de $\mathit{min}[H]$.

```
PISANO-ŞTERGE(H, x)
1: dacă x = min[H] atunci
2:    HEAP-FIB-EXTRAGE-MIN(H)
3: altfel
4:    y ← p[x]
5:    dacă y ≠ NIL atunci
6:       TAIE(H, x, y)
7:       TAIE-ÎN-CASCADĂ(H, y)
8:    adaugă lista de fii ai lui x la lista de rădăcini a lui H
9:    şterge x din lista de rădăcini a lui H
```

a. Afirmaţia profesorului privind execuţia mai rapidă se bazează în parte pe presupunerea că linia 8 poate fi executată într-un timp actual $O(1)$. Ce este greşit în această presupunere?

b. Daţi o margine superioară bună pentru timpul actual al operaţiei PISANO-ŞTERGE în cazul $x \neq \mathit{min}[H]$. Marginea găsită ar trebui exprimată în termeni de $\mathit{grad}[x]$ şi numărul c de apeluri al procedurii TĂIERE-ÎN-CASCADĂ.

c. Fie H' heap-ul Fibonacci rezultat după apelul PISANO-ŞTERGE(H, x). Presupunând că x nu este o rădăcină, estimaţi o margine pentru potenţialul lui H' în termeni de $\mathit{grad}[x]$, c, $t(H)$ şi $m(H)$.

d. Arătaţi că timpul amortizat pentru PISANO-ŞTERGE nu este asimptotic mai bun decât pentru HEAP-FIB-ŞTERGE, chiar dacă $x \neq \mathit{min}[H]$.

21-2 Alte operaţii pe heap-uri Fibonacci

Dorim să îmbunătăţim un heap Fibonacci H prin adăugarea a două operaţii fără a schimba timpul amortizat de execuţie al celorlalte operaţii pe heap-uri Fibonacci.

a. Daţi o implementare eficientă a operaţiei HEAP-FIB-SCHIMBĂ-CHEIE(H, x, k), care schimbă cheia nodului x atribuindu-i valoarea k. Analizaţi timpul amortizat de execuţie a implementării găsite pentru cazurile în care k este mai mare, mai mică, sau egală cu $\mathit{cheie}[x]$.

b. Daţi o implementare eficientă a operaţiei HEAP-FIB-TRUNCHIAZĂ(H, r), care şterge $\min(r, n[H])$ noduri din H. Nodurile care se şterg pot fi alese arbitrar. Analizaţi timpul amortizat de execuţie pentru implementarea găsită. (*Indicaţie:* Este posibil să fie necesară modificarea structurii de date sau a funcţiei potenţial.)

Note bibliografice

Heap-urile Fibonacci au fost introduse de Fredman şi Tarjan [75]. Articolul lor descrie, de asemenea, aplicarea heap-urilor Fibonacci problemelor de drum minim de sursă unică şi respectiv a tuturor perechilor, problemelor de potrivire bipartită ponderată şi problemei arborelui de acoperire minim.

În continuare Driscoll, Gabow, Shrairman şi Tarjan [58] au dezvoltat "heap-uri relaxate" ca o alternativă la heap-uri Fibonacci. Există două tipuri de heap-uri relaxate. Unul din ele oferă aceleaşi margini pentru timpul amortizat de execuţie ca şi heap-urile Fibonacci. Celălalt permite operaţiei DESCREŞTE-CHEIE, în cazul cel mai defavorabil (nu amortizat) o execuţie într-un timp $O(1)$, iar operaţiilor EXTRAGE-MIN şi ŞTERGE, în cazul cel mai defavorabil, o execuţie într-un timp $O(\lg n)$. În algoritmi paraleli heap-urile relaxate au de asemenea unele avantaje faţă de cele Fibonacci.

22 Structuri de date pentru mulţimi disjuncte

Unele aplicaţii implică gruparea a n elemente distincte într-o colecţie de mulţimi disjuncte. Pentru o astfel de colecţie există două operaţii importante: căutarea mulţimii căreia îi aparţine un element dat şi reuniunea a două mulţimi. Acest capitol propune metode pentru realizarea unei structuri de date care suportă aceste operaţii.

În secţiunea 22.1 se descriu operaţiile caracteristice unei structuri de date mulţimi disjuncte şi prezintă o aplicaţie simplă. În secţiunea 22.2, urmărim o implementare simplă pentru mulţimile disjuncte, bazată pe liste înlănţuite. O implementare mai eficientă folosind arbori cu rădăcină este prezentată în secţiunea 22.3. Timpul de execuţie, în cazul implementării bazate pe arbori cu rădăcină, este liniar pentru toate scopurile practice, dar teoretic este supraliniar. În secţiunea 22.4 se defineşte şi se analizează funcţia lui Ackermann şi inversa sa, care are o creştere foarte înceată; aceasta funcţie apare în evaluarea timpului de execuţie al operaţiilor, în cazul implementării bazate pe arbori. Se demonstrează apoi, folosind o analiză amortizată, existenţa unei margini superioare mai puţin exactă pentru timpul de execuţie.

22.1. Operaţii pe mulţimi disjuncte

O ***structură de date pentru mulţimi disjuncte*** memorează o colecţie $S = \{S_1, S_2, ..., S_k\}$ de mulţimi disjuncte dinamice. Fiecare mulţime este identificată printr-un ***reprezentant***, care este unul din elementele mulţimii. În anumite aplicaţii, nu are importanţă care membru este folosit ca reprezentant; ne interesează doar să-l alegem astfel încât dacă cerem reprezentantul unei mulţimi de două ori, fără să modificăm mulţimea între cele două cereri, să obţinem acelaşi răspuns de fiecare dată. În alte aplicaţii poate exista o regulă predefinită pentru alegerea reprezentantului, cum ar fi alegerea celui mai mic element din mulţime (presupunând bineînţeles că elementele pot fi ordonate).

Ca şi în celelalte implementări ale mulţimilor dinamice pe care le-am studiat, fiecare element al unei mulţimi este reprezentat printr-un obiect. Notând cu x un obiect, definim următoarele operaţii.

FORMEAZĂ-MULŢIME(x) creează o mulţime nouă cu un singur element x (care este şi reprezentant). Deoarece mulţimile sunt disjuncte, cerem ca x să nu fie deja în altă mulţime.

REUNEŞTE(x, y) reuneşte mulţimile dinamice S_x şi S_y, care conţin pe x, respectiv pe y, într-o nouă mulţime care este reuniunea celor două mulţimi. Se presupune că înaintea aplicării operaţiei cele două mulţimi sunt disjuncte. Reprezentatul mulţimii rezultate este un element oarecare al reuniunii $S_x \cup S_y$, deşi multe implementări ale operaţiei REUNEŞTE aleg fie reprezentantul lui S_x, fie al lui S_y, ca reprezentant al noii mulţimi. Pentru că noi cerem ca mulţimile din colecţie să fie disjuncte, "distrugem" mulţimile S_x şi S_y, ştergându-le din colecţie.

GĂSEŞTE-MULŢIME(x) returnează un pointer spre reprezentantul (unic) al mulţimii care-l conţine pe x.

Pe parcursul acestui capitol, vom analiza timpul de execuţie pentru structurile de date mulţimi disjuncte, în termenii a doi parametri: n, numărul de operaţii FORMEAZĂ-MULŢIME şi m, numărul total de operaţii FORMEAZĂ-MULŢIME, REUNEŞTE şi GĂSEŞTE-MULŢIME. Deoarece mulţimile sunt disjuncte, fiecare operaţie REUNEŞTE reduce numărul de mulţimi cu unu. Astfel după $n-1$ operaţii REUNEŞTE, rămâne o singură mulţime. Numărul de operaţii REUNEŞTE este deci cel mult $n-1$. Observaţi de asemenea că numărul total de operaţii m include operaţiile FORMEAZĂ-MULŢIME, deci vom avea $m \geq n$.

O aplicaţie a structurilor de date mulţimi disjuncte

Una dintre aplicaţiile structurilor de date mulţimi disjuncte este determinarea componentelor conexe ale unui graf neorientat (vezi secţiunea 5.4). De exemplu, figura 22.1(a) prezintă un graf cu patru componente conexe.

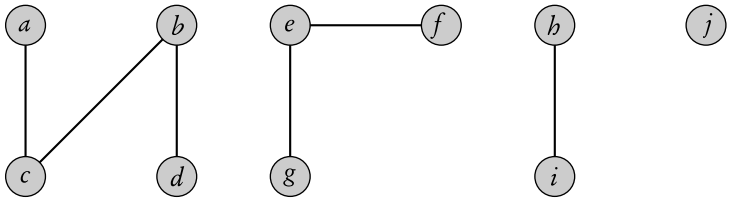
Următoarea procedură COMPONENTE-CONEXE foloseşte operaţiile pe mulţimi disjuncte pentru a determina componentele conexe ale unui graf. După ce COMPONENTE-CONEXE a fost executată ca un pas de preprocesare, procedura ACEEAŞI-COMPONENTĂ răspunde la cereri care verifică dacă două vârfuri sunt în aceeaşi componentă conexă.[1] (Mulţimea de vârfuri a unui graf G este notată cu $V[G]$ şi mulţimea muchiilor e notată cu $E[G]$.)

```
COMPONENTE-CONEXE(G)
1: pentru fiecare vârf v ∈ V[G] execută
2:    FORMEAZĂ-MULŢIME(v)
3: pentru fiecare muchie (u, v) ∈ E(G) execută
4:    dacă GĂSEŞTE-MULŢIME(u) ≠ GĂSEŞTE-MULŢIME (v) atunci
5:       REUNEŞTE(u, v)

ACEEAŞI-COMPONENTĂ(u, v)
1: dacă GĂSEŞTE-MULŢIME(u) = GĂSEŞTE-MULŢIME(v) atunci
2:    returnează ADEVĂRAT
3: altfel
4:    returnează FALS
```

Procedura COMPONENTE-CONEXE plasează iniţial fiecare vârf v într-o mulţime proprie. Apoi, pentru fiecare muchie (u, v), se reunesc mulţimile care conţin pe u şi pe v. Conform exerciţiului 22.1-2, după ce toate muchiile sunt procesate, două vârfuri sunt în aceeaşi componentă conexă dacă şi numai dacă obiectele corespunzătoare sunt în aceeaşi mulţime. Prin urmare, COMPONENTE-CONEXE calculează mulţimile astfel încât procedura ACEEAŞI-COMPONENTĂ poate determina dacă două vârfuri sunt în aceeaşi componentă conexă. Figura 22.1(b) ilustrează modul în care COMPONENTE-CONEXE calculează mulţimile disjuncte.

[1] Atunci când vârfurile unui graf sunt "statice" – nu se schimbă în timp – componentele conexe pot fi calculate mai rapid, folosind căutarea în adâncime (exerciţiul 23.3-9). Uneori, totuşi, muchiile sunt adăugate "dinamic" şi avem nevoie să menţinem componentele conexe pe măsură ce fiecare muchie este adăugată. În acest caz, implementarea prezentată aici poate fi mai eficientă decât execuţia din nou a căutării în adâncime, pentru fiecare muchie nouă.



(a)

Muchia procesată	Colecţie de mulţimi disjuncte									
mulţimea iniţială	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

(b)

Figura 22.1 **(a)** Un graf cu patru componente conexe: $\{a, b, c, d\}, \{e, f, g\}, \{h, i\}$ şi $\{j\}$. **(b)** Colecţia de mulţimi disjuncte după procesarea fiecărei muchii.

Exerciţii

22.1-1 Presupunem că procedura COMPONENTE-CONEXE se execută pentru graful neorientat $G = (V, E)$, unde $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ şi muchiile din E sunt procesate în următoarea ordine: $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d)$. Scrieţi vârfurile din fiecare componentă conexă după fiecare iteraţie a liniilor 3–5.

22.1-2 Arătaţi că după ce toate muchiile sunt procesate de către COMPONENTE-CONEXE, două vârfuri sunt în aceeaşi componentă conexă dacă şi numai dacă ele sunt în aceeaşi mulţime.

22.1-3 De câte ori este apelată procedura GĂSEŞTE-MULŢIME în timpul execuţiei procedurii COMPONENTE-CONEXE pe un graf neorientat $G = (V, E)$ având k componente conexe? De câte ori este apelată procedura REUNEŞTE? Exprimaţi răspunsurile în funcţie de $|V|, |E|$ şi k.

22.2. Reprezentarea mulţimilor disjuncte prin liste înlănţuite

Un mod simplu de a implementa o structură de date mulţimi disjuncte este de a reprezenta fiecare mulţime printr-o listă înlănţuită. Reprezentantul unei mulţimi se consideră a fi primul obiect din lista corespunzătoare acelei mulţimi. Fiecare obiect din lista înlănţuită conţine un element al mulţimii, un pointer spre un obiect care conţine următorul element al mulţimii şi un pointer înapoi spre reprezentant. Figura 22.2(a) prezintă două mulţimi. În interiorul fiecărei liste înlănţuite, obiectele pot apare în orice ordine (exceptând faptul că primul obiect din fiecare listă

este reprezentantul ei). Pentru fiecare listă se va reţine un pointer la primul obiect din listă şi un pointer spre ultimul obiect din listă.

În reprezentarea cu liste înlănţuite, atât FORMEAZĂ-MULŢIME cât şi GĂSEŞTE-MULŢIME sunt operaţii simple, care necesită un timp $O(1)$. Pentru a realiza operaţia FORMEAZĂ-MULŢIME(x), vom crea o nouă listă înlănţuită al cărei singur obiect este x. GĂSEŞTE-MULŢIME(x) returnează doar pointerul de la x înapoi la reprezentant.

O implementare simplă pentru reuniune

Cea mai simplă implementare pentru operaţia REUNEŞTE, folosind reprezentarea mulţimilor prin liste înlănţuite, ia un timp semnificativ mai mare decât FORMEAZĂ-MULŢIME sau GĂSEŞTE-MULŢIME. Aşa cum arată figura 22.2(b), realizăm REUNEŞTE(x, y) prin adăugarea listei lui x la sfârşitul listei lui y. Reprezentantul noii mulţimi este elementul care a fost iniţial reprezentantul mulţimii care îl conţine pe y. Din păcate trebuie să actualizăm pointerul spre reprezentant pentru fiecare obiect originar din lista lui x, operaţie care consumă un timp liniar în lungimea listei lui x.

Se ajunge uşor la o secvenţă de m operaţii care necesită un timp $\Theta(m^2)$. Fie $n = \lceil m/2 \rceil + 1$ şi $q = m - n + 1 = \lfloor m/2 \rfloor$ şi presupunem că avem obiectele $x_1, x_2, \ldots, x_n$. Apoi executăm secvenţa de $m = n + q - 1$ operaţii prezentate în figura 22.3. Este necesar un timp de $\Theta(n)$ pentru realizarea a n operaţii FORMEAZĂ-MULŢIME. Deoarece a i-a operaţie REUNEŞTE actualizează i obiecte, numărul total de obiecte actualizate de către toate operaţiile REUNEŞTE este

$$\sum_{i=1}^{q-1} i = \Theta(q^2).$$

În concluzie, timpul total cheltuit este $\Theta(n + q^2)$, care este $\Theta(m^2)$ deoarece $n = \Theta(m)$ şi $q = \Theta(m)$. Astfel, în medie, fiecare operaţie cere un timp $\Theta(m)$. Aceasta înseamnă că timpul amortizat de execuţie al unei operaţii este $\Theta(m)$.

O euristică a reuniunii ponderate

Implementarea de mai sus pentru procedura REUNEŞTE cere în medie, un timp $\Theta(m)$ pentru un apel, deoarece noi putem adăuga o listă mai lungă la o listă mai scurtă; trebuie să actualizăm pointerul spre reprezentant pentru fiecare obiect al listei lungi. Putem presupune că fiecare reprezentant include de asemenea şi lungimea listei sale (care e uşor de reţinut) şi că la fiecare reuniune se adaugă lista mai scurtă la lista mai lungă. În caz de egalitate adăugarea se face arbitrar. Cu această simplă ***euristică a reuniunii ponderate***, o operaţie REUNEŞTE poate să necesite totuşi un timp $\Omega(m)$, dacă ambele mulţimi au $\Omega(m)$ elemente. Totuşi aşa cum arată teorema următoare, o secvenţă de m operaţii FORMEAZĂ-MULŢIME, REUNEŞTE şi GĂSEŞTE-MULŢIME, dintre care n sunt operaţii FORMEAZĂ-MULŢIME, consumă un timp $O(m + n \lg n)$.

Teorema 22.1 Folosind reprezentarea cu liste înlănţuite pentru mulţimile disjuncte şi euristica reuniunii ponderate, o secvenţă de m operaţii FORMEAZĂ-MULŢIME, REUNEŞTE şi GĂSEŞTE-MULŢIME, dintre care n sunt operaţii FORMEAZĂ-MULŢIME, consumă un timp $O(m + n \lg n)$.

Demonstraţie. Începem prin a calcula, pentru fiecare obiect dintr-o mulţime de mărime n, o margine superioară pentru numărul de actualizări ale pointerului spre reprezentant al acelui

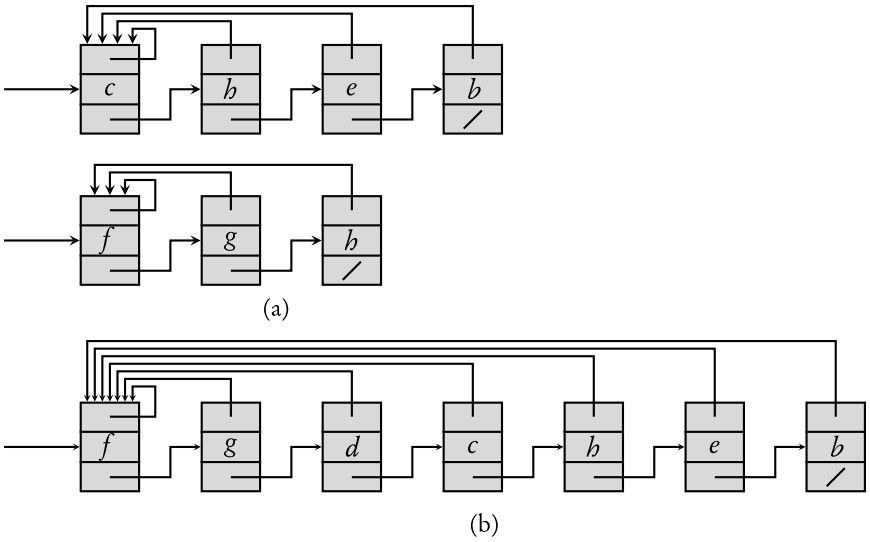


Figura 22.2 **(a)** Reprezentările prin liste înlănţuite a două mulţimi. Una conţine obiectele b, c, e şi h cu reprezentantul c şi cealaltă conţine obiectele d, f şi g, cu f ca reprezentant. Fiecare obiect din listă conţine un element al mulţimii, un pointer spre următorul obiect din listă şi un pointer înapoi spre primul obiect din listă, care este reprezentantul. **(b)** Rezultatul operaţiei REUNEŞTE(e, g). Reprezentantul mulţimii rezultate este f.

obiect. Considerăm un obiect fix x. Ştim că de fiecare dată când este actualizat pointerul spre reprezentant, trebuie ca x să aparţină mulţimii cu elemente mai puţine. Prima dată când se actualizează pointerul lui x spre reprezentant, mulţimea rezultat are cel puţin 2 elemente. Similar, a doua oară când pointerul lui x spre reprezentant este actualizat, mulţimea rezultată trebuie să aibă cel puţin 4 elemente. Continuând aşa, observăm că pentru orice $k \leq n$, după ce pointerul spre reprezentant al lui x a fost actualizat de $\lceil \lg k \rceil$ ori, mulţimea rezultat are cel puţin k elemente. Deoarece cea mai mare mulţime are cel mult n membri, rezultă că fiecare pointer spre reprezentant al unui obiect poate fi modificat de cel mult $\lceil \lg n \rceil$ ori în toate operaţiile REUNEŞTE. Timpul total folosit pentru a actualiza cele n obiecte este deci $O(n \lg n)$.

Timpul total pentru executarea întregii secvenţe de m operaţii se poate calcula uşor. Fiecare operaţie FORMEAZĂ-MULŢIME şi GĂSEŞTE-MULŢIME necesită un timp $O(1)$ şi sunt $O(m)$ operaţii de acest fel. Astfel, timpul total pentru întreaga secvenţă de operaţii este $O(m + n \lg n)$. ■

Exerciţii

22.2-1 Scrieţi procedurile FORMEAZĂ-MULŢIME, GĂSEŞTE-MULŢIME şi REUNEŞTE, folosind reprezentarea cu liste înlănţuite şi euristica reuniunii ponderate. Fiecare obiect x are atributele: $rep[x]$ care indică reprezentantul mulţimii care îl conţine pe x, $ultim[x]$ care indică ultimul obiect din lista înlănţuită care îl conţine pe x şi $dim[x]$ care ne dă cardinalul mulţimii care îl conţine pe x. Consideraţi că $ultim[x]$ şi $dim[x]$ sunt corecte doar dacă x este un reprezentant.

22.2-2 Determinaţi ce structură de date şi ce răspunsuri returnează operaţiile GĂSEŞTE-MULŢIME din următorul program. Folosiţi reprezentarea prin liste înlănţuite cu euristica

Operaţia	Număr de obiecte actualizate
FORMEAZĂ-MULŢIME(x_1)	1
FORMEAZĂ-MULŢIME(x_2)	1
$\vdots$	$\vdots$
FORMEAZĂ-MULŢIME(x_n)	1
REUNEŞTE(x_1, x_2)	1
REUNEŞTE(x_2, x_3)	2
REUNEŞTE(x_3, x_4)	3
$\vdots$	$\vdots$
REUNEŞTE(x_{q-1}, x_q)	$q-1$

Figura 22.3 O secvenţă de m operaţii care se execută într-un timp $\Theta(m^2)$, folosind reprezentarea cu liste înlănţuite şi implementarea directă pentru REUNEŞTE. Pentru acest exemplu $n = \lceil m/2 \rceil + 1$ şi $q = m - n$.

reuniunii ponderate.

```
1: pentru i ← 1, 16 execută
2:    FORMEAZĂ-MULŢIME(x_i)
3: pentru i ← 1, 15, 2 execută
4:    REUNEŞTE(x_i, x_{i+1})
5: pentru i ← 1, 13, 4 execută
6:    REUNEŞTE(x_i, x_{i+2})
7: REUNEŞTE(x_1, x_5)
8: REUNEŞTE(x_11, x_13)
9: REUNEŞTE(x_1, x_10)
10: GĂSEŞTE-MULŢIME(x_2)
11: GĂSEŞTE-MULŢIME(x_9)
```

22.2-3 Adaptaţi demonstraţia teoremei 22.1 pentru a obţine margini amortizate de timp $O(1)$ pentru FORMEAZĂ-MULŢIME şi GĂSEŞTE-MULŢIME şi $O(\lg n)$ pentru REUNEŞTE, folosind reprezentarea prin liste înlănţuite şi euristica reuniunii ponderate.

22.2-4 Daţi o margine asimptotică cât mai precisă pentru timpul de execuţie al secvenţei de operaţii din figura 22.3, presupunând o reprezentare prin liste înlănţuite şi euristica reuniunii ponderate.

22.3. Păduri de mulţimi disjuncte

Într-o implementare mai rapidă a mulţimilor disjuncte, reprezentăm mulţimile prin arbori cu rădăcină, în care fiecare nod conţine un membru şi fiecare arbore reprezintă o mulţime. Într-o ***pădure de mulţimi disjuncte***, ilustrată în figura 22.4(a), fiecare membru indică doar spre părintele lui. Rădăcina fiecărui arbore conţine reprezentantul mulţimii, care este propriul său

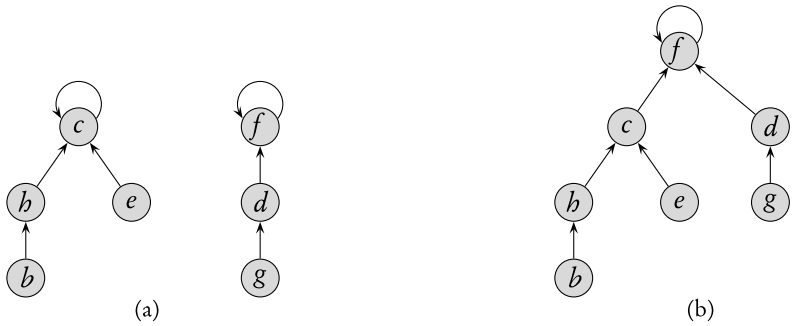


Figura 22.4 O pădure de mulţimi disjuncte. **(a)** Doi arbori care reprezintă cele două mulţimi din figura 22.2. Arborele din stânga reprezintă mulţimea $\{b, c, e, h\}$ cu c ca reprezentant, iar arborele din dreapta reprezintă mulţimea $\{d, f, g\}$ cu reprezentantul f. **(b)** Rezultatul operaţiei REUNEŞTE(e, g).

părinte. După cum vom vedea, deşi algoritmii direcţi care folosesc această reprezentare nu sunt mai rapizi decât cei care folosesc reprezentarea cu liste înlănţuite, prin introducerea a două euristici – "reuniune după rang" şi "comprimarea drumului" – putem obţine structura de date mulţimi disjuncte cea mai rapidă asimptotic, din cele cunoscute.

Vom realiza cele trei operaţii pe mulţimi disjuncte după cum urmează. Operaţia FORMEAZĂ-MULŢIME creează un arbore cu un singur nod. Operaţia GĂSEŞTE-MULŢIME o executăm prin urmărirea pointerilor spre părinţi până când găsim rădăcina arborelui. Nodurile vizitate pe drumul spre rădăcină constituie ***drumul de căutare***. O operaţie REUNEŞTE va avea efectul că rădăcina unui arbore va pointa spre rădăcina celuilalt arbore, aşa cum se arată în figura 22.4(b).

Euristici care îmbunătăţesc timpul de execuţie

Până acum nu am îmbunătăţit implementarea cu liste înlănţuite. O secvenţă de $n-1$ operaţii REUNEŞTE poate crea un arbore care este un lanţ liniar de n noduri. Totuşi, prin folosirea a două euristici, putem obţine un timp de execuţie care este aproape liniar în raport cu numărul total de operaţii m.

Prima euristică, ***reuniune după rang***, este similară cu euristica reuniune ponderată pe care am folosit-o la reprezentarea cu liste înlănţuite. Ideea este ca rădăcina arborelui cu mai puţine noduri să indice spre rădăcina arborelui cu mai multe noduri. În loc de a păstra explicit dimensiunea subarborelui cu rădăcină în fiecare nod, vom folosi o abordare care uşurează analiza. Pentru fiecare nod vom menţine un ***rang*** care aproximează logaritmul dimensiunii subarborelui şi care este de asemenea o margine superioară a înălţimii nodului. Folosind reuniunea după rang în timpul operaţiei REUNEŞTE, rădăcina cu rangul cel mai mic va indica spre rădăcina cu rang mai mare.

A doua euristică, ***comprimarea drumului***, este de asemenea foarte simplă şi foarte eficientă. Aşa cum arată figura 22.5, o folosim în timpul operaţiei GĂSEŞTE-MULŢIME, pentru ca fiecare nod de pe drumul de căutare să pointeze direct spre rădăcină. Comprimarea drumului nu modifică nici un rang.

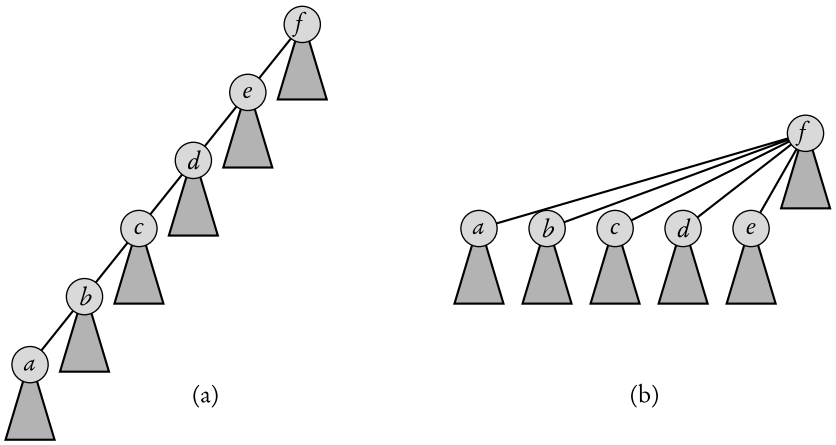


Figura 22.5 Comprimarea drumului în timpul operaţiei GĂSEŞTE-MULŢIME. Sunt omise săgeţile şi buclele. **(a)** Un arbore care reprezintă o mulţime, înaintea operaţiei GĂSEŞTE-MULŢIME(a). Triunghiurile reprezintă subarbori ale căror rădăcini sunt nodurile afişate. Fiecare nod are un pointer spre părintele său. **(b)** Aceeaşi mulţime după executarea operaţiei GĂSEŞTE-MULŢIME(a). Fiecare nod de pe drumul de căutare pointează acum direct spre rădăcină.

Algoritmi (în pseudocod) pentru păduri de mulţimi disjuncte

Pentru a implementa o pădure de mulţimi disjuncte cu euristica reuniunii după rang, trebuie să păstrăm valorile rangurilor. Pentru fiecare nod x, reţinem valoarea întregă $\mathit{rang}[x]$, care este o margine superioară a înălţimii lui x (numărul de muchii al celui mai lung drum de la x la o frunză descendentă). Atunci când FORMEAZĂ-MULŢIME creează o mulţime cu un element, rangul iniţial al singurului nod în arborele corespunzător este 0. Fiecare operaţie GĂSEŞTE-MULŢIME lasă toate rangurile neschimbate. În urma aplicării operaţiei REUNEŞTE pentru doi arbori, transformăm rădăcina cu rang mai mare în părinte al rădăcinii cu rang mai mic. În caz de egalitate, alegem arbitrar una dintre rădăcini şi incrementăm rangul său.

Vom scrie această metodă în pseudocod. Reţinem părintele nodului x în $p[x]$. REUNEŞTE apelează o subrutină, procedura UNEŞTE, care are parametrii de intrare pointeri spre două rădăcini.

Procedura GĂSEŞTE-MULŢIME este o ***metodă în două treceri***: face un pas în sus pe drumul de căutare pentru a găsi rădăcina şi face al doilea pas în jos pe drumul de căutare pentru a actualiza fiecare nod astfel încât să pointeze direct spre rădăcină. Fiecare apel al procedurii GĂSEŞTE-MULŢIME(x) returnează $p[x]$ în linia 3. Dacă x este rădăcină, atunci linia 2 nu este executată şi este returnată valoarea $p[x] = x$. În acest moment recurenţa coboară. Altfel, linia 2 este executată şi apelul recursiv cu parametrul $p[x]$ returnează un pointer spre rădăcină. Linia 2 actualizează nodul x astfel încât să indice direct spre rădăcină şi acest pointer este returnat în linia 3.

FORMEAZĂ-MULŢIME(x)
1: $p[x] \leftarrow x$
2: $\mathit{rang}[x] \leftarrow 0$

```
REUNEŞTE(x, y)
1: UNEŞTE(GĂSEŞTE-MULŢIME(x),GĂSEŞTE-MULŢIME(y))

UNEŞTE(x, y)
1: dacă rang[x] > rang[y] atunci
2:    p[y] ← x
3: altfel
4:    p[x] ← y
5:    dacă rang[x] = rang[y] atunci
6:       rang[y] ← rang[y] + 1
```

Procedura GĂSEŞTE-MULŢIME cu comprimarea drumului este simplă.

```
GĂSEŞTE-MULŢIME(x)
1: dacă x ≠ p[x] atunci
2:    p[x] ←GĂSEŞTE-MULŢIME(p[x])
3: returnează p[x]
```

Efectul euristicilor asupra timpului de execuţie

Fiecare euristică separat, atât reuniunea după rang cât şi comprimarea drumului, îmbunătăţeste timpul de execuţie al operaţiilor pe păduri de mulţimi disjuncte şi această îmbunătăţire este chiar mai mare atunci când sunt folosite împreună. Numai reuniunea după rang conduce la acelaşi timp de execuţie pe care l-am obţinut folosind euristica reuniunii ponderate pentru reprezentarea cu liste înlănţuite: implementarea rezultată se execută într-un timp $O(m \lg n)$ (vezi exerciţiul 22.4-3). Această mărginire este strictă (vezi exerciţiul 22.3-3). Deşi nu vom demonstra acest fapt, dacă există n operaţii FORMEAZĂ-MULŢIME (şi deci cel mult $n-1$ operaţii REUNEŞTE) şi f operaţii GĂSEŞTE-MULŢIME, euristica compresiei drumului, singură, ne dă un timp de execuţie, în cazul cel mai defavorabil, de $\Theta(f \log_{(1+f/n)} n)$, dacă $f \geq n$ şi $\Theta(n + f \lg n)$ dacă $f < n$.

Atunci când folosim atât reuniunea după rang cât şi comprimarea drumului, timpul de execuţie în cazul cel mai defavorabil este $O(m\alpha(m, n))$, unde $\alpha(m, n)$ este inversa funcţiei lui Ackermann, cu o creştere *foarte* înceată, definită în secţiunea 22.4. Aplicaţiile structurii de date mulţimi disjuncte, care pot fi luate în considerare, satisfac $\alpha(m, n) \leq 4$; astfel putem considera timpul de execuţie ca fiind liniar în m, în toate situaţiile practice. În secţiunea 22.4, construim o margine mai puţin precisă de $O(m \lg^* n)$.

Exerciţii

22.3-1 Rezolvaţi exerciţiul 22.2-2, folosind o pădure de mulţimi disjuncte cu euristicile reuniune după rang şi comprimarea drumului.

22.3-2 Scrieţi o versiune nerecursivă pentru GĂSEŞTE-MULŢIME cu comprimarea drumului.

22.3-3 Daţi o secvenţă de m operaţii FORMEAZĂ-MULŢIME, REUNEŞTE şi GĂSEŞTE-MULŢIME, dintre care n sunt operaţii FORMEAZĂ-MULŢIME, care se execută într-un timp $\Omega(m \lg n)$, dacă se foloseşte doar reuniunea după rang.

	$j=1$	$j=2$	$j=3$	$j=4$
$i=1$	2^1	2^2	2^2	2^2
$i=2$	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i=3$	2^{2^2}	$2^{2^{\cdot^{\cdot^{\cdot^2}}}\}16}$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}\}16}}$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}\}16}}}$

Figura 22.6 Valorile lui $A(i, j)$ pentru valori mici ale lui i şi j.

22.3-4 $\star$ Arătaţi că orice secvenţă de m operaţii FORMEAZĂ-MULŢIME, GĂSEŞTE-MULŢIME şi UNEŞTE, în care toate operaţiile UNEŞTE apar înaintea oricărei operaţii GĂSEŞTE-MULŢIME, se execută doar într-un timp $O(m)$, dacă se foloseşte atât reuniunea după rang cât şi comprimarea drumului. Ce se întâmplă în aceeaşi situaţie dacă se foloseşte doar euristica comprimării drumului?

22.4. Analiza reuniunii după rang comprimând drumul

Aşa cum am observat în secţiunea 22.3, pentru m operaţii pe mulţimi disjuncte având n elemente, timpul de execuţie, în cazul folosirii euristicii combinate reuniune după rang şi comprimarea drumului, este $O(m\alpha(m, n))$. În această secţiune vom analiza funcţia α pentru a vedea cât de încet creşte. Apoi, vom da o demonstraţie pentru o margine superioară mai puţin exactă de $O(m \lg^* n)$, care este puţin mai simplă decât demonstraţia foarte complexă pentru timpul de execuţie $O(m\alpha(m, n))$.

Funcţia lui Ackermann şi inversa sa

Pentru a înţelege funcţia lui Ackermann şi inversa sa α, introducem o notaţie pentru exponenţierea repetată. Dacă $i \geq 0$ este un număr întreg, notaţia

$$2^{2^{\cdot^{\cdot^{\cdot^2}}}\}i}$$

se foloseşte pentru funcţia $g(i)$ definită recursiv astfel

$$g(i) = \begin{cases} 2^1 & \text{dacă } i = 0, \\ 2^2 & \text{dacă } i = 1, \\ 2^{g(i-1)} & \text{dacă } i > 1. \end{cases}$$

Parametrul i exprimă intuitiv "înălţimea stivei de 2" care reprezintă exponentul. De exemplu

$$2^{2^{\cdot^{\cdot^{\cdot^2}}}\}4} = 2^{2^{2^{2^2}}} = 2^{65536}.$$

Reamintim definiţia funcţiei $\lg^*$ în termenii funcţiilor $\lg^{(i)}$ definite pentru orice număr întreg $i \geq 0$:

$$\lg^{(i)} n = \begin{cases} n & \text{dacă } i = 0, \\ \lg(\lg^{(i-1)} n) & \text{dacă } i > 0 \text{ şi } \lg^{(i-1)} n > 0, \\ \text{nedefinit} & \text{dacă } i > 0 \text{ şi } \lg^{(i-1)} n \leq 0 \text{ sau } \lg^{(i-1)} n \text{ este nedefinit ;} \end{cases}$$

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}.$$

Funcţia $\lg^*$ este în esenţă inversa exponenţierii repetate:

$$\lg^* 2^{\overset{2}{\cdot^{\cdot^{\cdot}}}\}n} = n + 1.$$

Putem acum să scriem funcţia lui Ackermann, care e definită pentru întregii $i, j \geq 1$ prin

$$\begin{array}{lcll} A(1,j) & = & 2^j & \text{pentru } j \geq 1, \\ A(i,1) & = & A(i-1,2) & \text{pentru } i \geq 2, \\ A(i,j) & = & A(i-1, A(i,j-1)) & \text{pentru } i,j \geq 2. \end{array}$$

Figura 22.6 conţine valorile funcţiei pentru valori mici ale lui i şi j.

Figura 22.7 ilustrează într-un mod sistematic, de ce funcţia lui Ackermann are o creştere atât de rapidă. Prima linie, exponenţială în numărul de coloane j, are deja o creştere rapidă. A două linie constă din submulţimea "spaţiată" de coloane $2, 2^2, 2^{2^2}, 2^{2^{2^2}}, \ldots$ din prima linie. Segmentele dintre liniile adiacente indică acele coloane din linia cu număr mai mic care aparţin submulţimii descrise în linia cu număr mai mare. A treia linie constă dintr-o submulţime de coloane şi mai larg "spaţiată" $2, 2^{2^2}, 2^{\overset{2}{\cdot^{\cdot^{\cdot}}}\}16}, 2^{\overset{2}{\cdot^{\cdot^{\cdot}}}\}2^{\overset{2}{\cdot^{\cdot^{\cdot}}}\}16}}, \ldots$ din a doua linie şi este la rândul ei o submulţime şi mai rară de coloane din prima linie . În general spaţierea între coloanele liniei $i-1$ care apar în linia i creşte dramatic atât în numărul de coloane cât şi în numărul de linii. Observaţi că $A(2,j) = 2^{\overset{2}{\cdot^{\cdot^{\cdot}}}\}j}$ pentru orice număr întreg $j \geq 1$. Astfel, pentru $i > 2$, funcţia $A(i,j)$ creşte chiar mai rapid decât $2^{\overset{2}{\cdot^{\cdot^{\cdot}}}\}j}$.

Definim inversa funcţiei Ackermann prin[2]

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) > \lg n\}.$$

Dacă fixăm valoarea lui n, atunci pe măsură ce m creşte, funcţia $\alpha(m,n)$ este monoton descrescătoare. Pentru a înţelege această proprietate, observaţi că $\lfloor m/n \rfloor$ este monoton crescătoare pentru m crescător; de aici, deoarece n este fix, cea mai mică valoare a lui i pentru care $A(i, \lfloor m/n \rfloor)$ este mai mai mare decât $\lg n$ este monoton descrescătoare. Această proprietate corespunde intuitiv pădurilor de mulţimi disjuncte cu comprimarea drumului: pentru un număr dat de n elemente distincte, dacă numărul de operaţii m creşte, ne aşteptăm ca lungimea drumului mediu de căutare să scadă datorită comprimării drumului. Dacă realizăm m operaţii într-un timp $O(m\alpha(m,n))$,

[2] Deşi această funcţie nu este inversa funcţiei lui Ackermann în sens matematic, totuşi această funcţie are calităţile inversei în privinţa creşterii, care este pe atât de înceată pe cât de rapidă este creşterea funcţiei lui Ackermann. Motivul pentru care se foloseşte pragul surprinzător $\lg n$ este relevat în demonstraţia pentru timpul de execuţie $O(m\alpha(m,n))$, care este în afara scopului acestei cărţi.

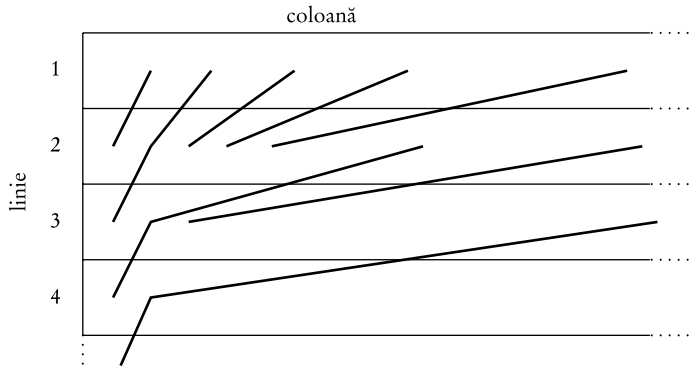


Figura 22.7 Creşterea explozivă a funcţiei lui Ackermann. Segmentele dintre liniile $i-1$ şi i indică intrări din linia $i-1$ în linia i. Datorită creşterii explozive, spaţierea orizontală nu poate fi scalată. Spaţierile orizontale dintre intrările liniei $i-1$ apar în linia i mult mărite pe măsură ce creşte numărul de linie şi coloană. Creşterea este chiar mai evidentă dacă urmărim intrările liniei i de la apariţia lor originală în linia 1.

atunci timpul mediu pentru o operaţie este $O(\alpha(m,n))$, care este monoton descrescător, atunci când m creşte.

Pentru a ne întoarce la afirmaţia că $\alpha(m,n) \leq 4$, în toate cazurile practice, observăm mai întâi că valoarea $\lfloor m/n \rfloor$ este cel puţin 1, deoarece $m \geq n$. Pentru că funcţia lui Ackermann este strict crescătoare în fiecare argument, $\lfloor m/n \rfloor \geq 1$ implică $A(i, \lfloor m/n \rfloor) \geq A(i,1)$ pentru orice $i \geq 1$. În particular, $A(4, \lfloor m/n \rfloor) \geq A(4,1)$. Dar avem de asemenea că

$$A(4,1) = A(3,2) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}\}16}},$$

care este cu mult mai mare decât numărul estimat de atomi, din universul observabil (aproximativ 10^{80}). Inegalitatea are loc doar pentru valori ale lui n impracticabil de mari, pentru care $A(4,1) \leq \lg n$ şi astfel $\alpha(m,n) \leq 4$, în toate cazurile practice. Observaţi că diferenţa de precizie dintre marginea $O(m \lg^* n)$ şi marginea $O(m\alpha(m,n))$ este foarte mică; $\lg^* 65536 = 4$ şi $\lg^* 2^{65536} = 5$, deci $\lg^* n \leq 5$ în toate cazurile practice.

Proprietăţile rangurilor

Demonstrăm în continuarea acestei secţiuni că $O(m \lg^* n)$ reprezintă o margine superioară a timpului de execuţie a operaţiilor pe mulţimi disjuncte cu reuniune după rang şi comprimarea drumului. Pentru a demonstra acest lucru, demonstrăm mai întâi câteva proprietăţi simple ale rangurilor.

Lema 22.2 Pentru toate nodurile x, avem $\mathit{rang}[x] \leq \mathit{rang}[p[x]]$ cu inegalitate strictă dacă $x \neq p[x]$. Valoarea $\mathit{rang}[x]$ iniţial este 0 şi creşte în timp până ce $x \neq p[x]$; după aceea $\mathit{rang}[x]$ nu se mai modifică. Valoarea $\mathit{rang}[p[x]]$ este o funcţie monoton crescătoare în timp.

Demonstraţie. Demonstraţia este o inducţie directă după numărul de operaţii, folosind implementările operaţiilor FORMEAZĂ-MULŢIME, REUNEŞTE şi GĂSEŞTE-MULŢIME, care apar în secţiunea 22.3. Este propusă spre rezolvare în exerciţiul 22.4-1. ■

Definim $\dim(x)$ ca fiind numărul de noduri din arborele cu rădăcina x, inclusiv nodul x.

Lema 22.3 Pentru toate rădăcinile x de arbori, $\dim(x) \geq 2^{rang[x]}$.

Demonstraţie. Demonstraţia se face prin inducţie după numărul de operaţii UNEŞTE. Observaţi că operaţiile GĂSEŞTE-MULŢIME nu schimbă nici rangul unei rădăcini al unui arbore şi nici dimensiunea arborelui său.

Baza: Lema este adevărată înainte de prima operaţie UNEŞTE, deoarece rangurile sunt iniţial 0 şi fiecare arbore conţine cel puţin un nod.

Pasul inductiv: Presupunem că lema este adevărată înaintea execuţiei operaţiei UNEŞTE(x, y). Fie *rang* rangul chiar de dinainte de UNEŞTE şi fie $rang'$ rangul chiar după UNEŞTE. Definim similar *dim* şi dim'.

Dacă *rang*$[x] \neq$ *rang*$[y]$, presupunem fără să pierdem generalitatea că *rang*$[x] <$ *rang*$[y]$. Nodul y este rădăcina arborelui format de operaţia UNEŞTE şi

$$\dim'(y) = \dim(x) + \dim(y) \geq 2^{rang[x]} + 2^{rang[y]} \geq 2^{rang[y]} = 2^{rang'[y]}.$$

În afară de nodul y nici un alt nod nu îşi schimbă valorile pentru rang şi dimensiune.

Dacă *rang*$[x] =$ *rang*$[y]$, nodul y este din nou rădăcina unui nou arbore şi

$$\dim'(y) = \dim(x) + \dim(y) \geq 2^{rang[x]} + 2^{rang[y]} = 2^{rang[y]+1} = 2^{rang'[y]}.$$

■

Lema 22.4 Pentru orice întreg $r \geq 0$ există cel mult $n/2^r$ noduri de rang r.

Demonstraţie. Fixăm o valoare particulară pentru r. Presupunem că atunci când atribuim un rang r unui nod x (în linia 2 din FORMEAZĂ-MULŢIME sau în linia 5 din UNEŞTE), ataşăm o etichetă x fiecărui nod din arborele cu rădăcina x. Conform lemei 22.3, cel puţin 2^r noduri sunt etichetate de fiecare dată. Presupunem că rădăcina arborelui care conţine nodul x se schimbă. Lema 22.2 ne asigură că rangul noii rădăcini (sau de fapt, al fiecărui strămoş al lui x) este cel puţin $r + 1$. Deoarece ataşăm etichete doar atunci când unei rădăcini i se atribuie rangul r, nici un nod din acest arbore nou nu va mai fi etichetat din nou. Astfel fiecare nod este etichetat cel mult o dată, atunci când rădăcinii lui îi este atribuit rangul r. Pentru că sunt n noduri, sunt cel mult n noduri etichetate, cu cel puţin 2^r etichete ataşate pentru fiecare nod de rang r. Dacă au fost mai mult decât $n/2^r$ noduri de rang r, atunci vor fi etichetate mai mult decât $2^r \cdot (n/2^r) = n$ noduri de către un nod de rang r, ceea ce este o contradicţie. Rezultă că cel mult $n/2^r$ noduri vor avea rangul r. ■

Corolarul 22.5 Orice nod are rangul cel mult $\lfloor \lg n \rfloor$.

Demonstraţie. Dacă presupunem că $r > \lg n$, atunci sunt cel mult $n/2^r < 1$ noduri cu rangul r. Deoarece rangurile sunt numere naturale, corolarul este demonstrat. ■

Demonstrarea marginii de timp

Vom folosi o metodă de analiză amortizată (vezi secţiunea 18.1) pentru a demonstra marginea de timp de $O(m \lg^* n)$. Pentru realizarea analizei amortizate este convenabil să presupunem că se foloseşte mai degrabă operaţia UNEŞTE, decât operaţia REUNEŞTE. Aceasta presupune execuţia,

dacă este necesar, a operaţiilor GĂSEŞTE-MULŢIME corespunzătoare, deoarece parametrii procedurii UNEŞTE sunt pointeri spre două rădăcini. Lema următoare arată că şi dacă numărăm operaţiile GĂSEŞTE-MULŢIME suplimentare, timpul asimptotic de execuţie rămîne neschimbat.

Lema 22.6 Presupunem că transformăm o secvenţă S' de m' operaţii FORMEAZĂ-MULŢIME, REUNEŞTE şi GĂSEŞTE-MULŢIME într-o secvenţă S de m operaţii FORMEAZĂ-MULŢIME, UNEŞTE şi GĂSEŞTE-MULŢIME prin înlocuirea fiecărei operaţii REUNEŞTE cu două operaţii GĂSEŞTE-MULŢIME urmate de o operaţie UNEŞTE. Dacă secvenţa S se execută într-un timp $O(m \lg^* n)$, atunci secvenţa S' se execută într-un timp $O(m' \lg^* n)$.

Demonstraţie. Deoarece fiecare operaţie REUNEŞTE din secvenţa S' este convertită în trei operaţii în S, avem $m' \leq m \leq 3m'$. Pentru că $m = O(m')$, o margine de timp de $O(m \lg^* n)$ pentru secvenţa convertită S implică o margine de timp $O(m' \lg^* n)$ pentru secvenţa originală S'. ■

În continuare, în această secţiune, vom presupune că secvenţa iniţială de m' operaţii FORMEAZĂ-MULŢIME, REUNEŞTE şi GĂSEŞTE-MULŢIME a fost convertită într-o secvenţă de m operaţii FORMEAZĂ-MULŢIME, UNEŞTE şi GĂSEŞTE-MULŢIME. Vom demonstra acum o margine de timp de $O(m \lg^* n)$ pentru secvenţa convertită şi vom apela la lema 22.6 pentru a demonstra timpul de execuţie de $O(m' \lg^* n)$ pentru secvenţa originală cu m' operaţii.

Teorema 22.7 O secvenţă de m operaţii FORMEAZĂ-MULŢIME, UNEŞTE şi GĂSEŞTE-MULŢIME, dintre care n sunt operaţii FORMEAZĂ-MULŢIME, poate fi executată pe o pădure de mulţimi disjuncte cu reuniune după rang şi cu comprimarea drumului într-un timp $O(m \lg^* n)$, în cazul cel mai defavorabil.

Demonstraţie. Atribuim ***ponderile*** corespunzătoare costului real pentru fiecare operaţie pe mulţime şi calculăm numărul total de ponderi atribuite după ce întreaga secvenţă de operaţii a fost executată. Acesta ne va da costul real pentru toate operaţiile pe mulţime.

Ponderile atribuite pentru operaţiile FORMEAZĂ-MULŢIME şi UNEŞTE sunt simple: o pondere pe operaţie. Pentru că fiecare operaţie necesită un timp real de $O(1)$, ponderile atribuite sunt egale cu costurile reale ale operaţiilor.

Înainte de a evalua ponderile atribuite pentru operaţiile GĂSEŞTE-MULŢIME, partiţionăm rangurile nodurilor în ***blocuri*** prin introducerea rangului r în blocul $\lg^* r$ pentru $r = 0, 1, ..., \lfloor \lg n \rfloor$. (Reamintim că $\lfloor \lg n \rfloor$ este rangul maxim.) Deci blocul cu număr maxim este blocul $\lg^*(\lg n) = \lg^* n - 1$. Pentru simplificarea notării, definim pentru numerele întregi $j \geq -1$,

$$B(j) = \begin{cases} -1 & \text{dacă } j = -1, \\ 1 & \text{dacă } j = 0, \\ 2 & \text{dacă } j = 1, \\ 2^{2^{\cdot^{\cdot^{2}}}}\Big\}_{j-1} & \text{dacă } j \geq 2. \end{cases}$$

Apoi, pentru $j = 0, 1, ..., \lg^* n - 1$, al j-lea bloc constă din mulţimea rangurilor

$$\{B(j-1) + 1, B(j-1) + 2, ..., B(j)\}.$$

Folosim două tipuri de ponderi pentru o operaţie GĂSEŞTE-MULŢIME: ***ponderi bloc*** şi ***ponderi drum***. Presupunem că GĂSEŞTE-MULŢIME începe la nodul x_0 şi că drumul de căutare

constă din nodurile $x_0, x_1, ..., x_l$, unde pentru $i = 1, 2, \ldots, l$, nodul x_i este $p[x_{i-1}]$ şi x_l(o rădăcină) este $p[x_l]$. Pentru $j = 0, 1, \ldots, \lg^* n - 1$ atribuim o pondere bloc pentru *ultimul* nod din drum cu rangul în blocul j. (Observaţi că din lema 22.2 se deduce că pe orice drum de căutare, nodurile cu rangul într-un anumit bloc dat sunt consecutive.) Atribuim de asemenea o pondere bloc pentru fiul rădăcinii, care este x_{l-1}. Deoarece rangurile sunt strict crescătoare de-a lungul oricărui drum de căutare, o formulare echivalentă stabileşte o pondere bloc fiecărui nod x_i astfel încât $p[x_i] = x_l$ (x_i este rădăcina sau fiul ei) sau $\lg^* \mathit{rang}[x_i] < \lg^* \mathit{rang}[x_{i+1}]$ (blocul cu rangul lui x_i diferă de cel al părintelui său). Fiecărui nod de pe drumul de căutare căruia nu îi stabilim o pondere bloc, îi stabilim o pondere drum.

O dată ce unui nod, altul decât rădăcina sau fiul ei, îi sunt stabilite ponderi bloc, nu-i vor mai fi stabilite mai departe ponderi drum. Pentru a vedea motivul să observăm că de fiecare dată când apare o comprimare de drum, rangul unui nod x_i pentru care $p[x_i] \neq x_l$, rămâne acelaşi, dar noul părinte al lui x_i are rangul strict mai mare decât rangul vechiului părinte al lui x_i. Diferenţa dintre rangurile lui x_i şi al părintelui său este o funcţie monoton crescătoare în timp. Astfel, diferenţa dintre $\lg^* \mathit{rang}[p[x_i]]$ şi $\lg^* \mathit{rang}[x_i]$ este deasemenea o funcţie monoton crescătoare în timp. O dată ce x_i şi părintele lui ajung să aibă rangurile în blocuri diferite, vor avea mereu rangurile în blocuri diferite şi astfel lui x_i nu i se va mai asocia o pondere drum.

Deoarece am atribuit pondere o dată pentru fiecare nod vizitat în fiecare operaţie GĂSEŞTE-MULŢIME, numărul total de ponderi stabilite este numărul total de noduri vizitate în toate operaţiile GĂSEŞTE-MULŢIME; acest total reprezintă costul real pentru toate operaţiile GĂSEŞTE-MULŢIME. Vrem să arătăm că acest total este $O(m \lg^* n)$.

Numărul de ponderi bloc este uşor de mărginit. Există cel mult o pondere bloc asociată fiecărui număr de bloc de pe drumul de căutare dat, plus o pondere bloc pentru fiul rădăcinii. Deoarece numerele de bloc aparţin intervalului de la 0 la $\lg^* n - 1$, sunt cel mult $\lg^* n + 1$ ponderi de bloc pentru fiecare operaţie GĂSEŞTE-MULŢIME. Astfel, sunt cel mult $m(\lg^* n + 1)$ ponderi bloc asociate în toate operaţiile GĂSEŞTE-MULŢIME.

Mărginirea ponderilor drum este mai puţin directă. Începem prin a observa că dacă nodului x_i îi este asociată o pondere drum, atunci $p[x_i] \neq x_l$ înaintea comprimării drumului, deci lui x_i i se va atribui un părinte nou în timpul comprimării drumului. Mai mult, aşa cum am observat înainte, noul părinte al lui x_i are rangul mai mare decât vechiul părinte. Presupunem că rangul nodului x_i este în blocul j. De câte ori i se poate atribui lui x_i un nou părinte, şi astfel să i se asocieze o pondere drum, înainte ca să i se atribuie un părinte al cărui rang este într-un bloc diferit (după care nu i se va mai asocia vreo pondere drum)? Acest număr este maxim atunci când x_i are cel mai mic rang din blocul său, şi anume $B(j-1)+1$ şi rangurile părinţilor săi succesivi iau valorile $B(j-1)+2, B(j-1)+3, ..., B(j)$. Deoarece sunt $B(j) - B(j-1) - 1$ astfel de ranguri, deducem că unui vârf i se pot asocia cel mult $B(j) - B(j-1) - 1$ ponderi drum, atâta timp cât rangul său este în blocul j.

Pasul următor în mărginirea ponderilor drum este mărginirea numărului de noduri care au rangurile în blocul j pentru $j \geq 0$. (Reamintim că, pe baza lemei 22.2 că un nod nu-şi mai schimbă rangul după ce devine fiul altui nod.) Notăm cu $N(j)$ numărul de noduri ale căror ranguri sunt în blocul j. Apoi, din lema 22.4 avem

$$N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r}.$$

Pentru $j = 0$, această sumă se evaluează la

$$N(0) = n/2^0 + n/2^1 = 3n/2 = 3n/2B(0).$$

Pentru $j \geq 1$ avem

$$N(j) \leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r} < \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} = \frac{n}{2^{B(j-1)}} = \frac{n}{B(j)}.$$

Astfel, $N(j) \leq n/B(j)$ pentru orice număr întreg $j \geq 0$.

Încheiem mărginirea ponderilor drum prin însumarea peste blocuri a produsului dintre numărul maxim de noduri cu rangurile într-un bloc şi numărul maxim de ponderi drum pentru un nod din acel bloc. Notând cu $P(n)$ numărul total de ponderi drum avem

$$P(n) \leq \sum_{j=0}^{\lg^* n - 1} \frac{n}{B(j)}(B(j) - B(j-1) - 1) \leq \sum_{j=0}^{\lg^* n - 1} \frac{n}{B(j)} \cdot B(j) = n \lg^* n.$$

Deci numărul total de ponderi impuse de operaţiile GĂSEŞTE-MULŢIME este $O(m(\lg^* n + 1) + n \lg^* n)$, care este $O(m \lg^* n)$ deoarece $m \geq n$. Deoarece avem $O(n)$ operaţii FORMEAZĂ-MULŢIME şi UNEŞTE, cu câte o pondere fiecare, timpul total este $O(m \lg^* n)$. ∎

Corolarul 22.8 O secvenţă de m operaţii FORMEAZĂ-MULŢIME, REUNEŞTE şi GĂSEŞTE-MULŢIME, dintre care n sunt operaţii FORMEAZĂ-MULŢIME, pot fi executate pe o pădure de mulţimi disjuncte cu reuniune după rang şi comprimarea drumului într-un timp $O(m \lg^* n)$, în cazul cel mai defavorabil.

Demonstraţie. Este imediată din teorema 22.7 şi lema 22.6. ∎

Exerciţii

22.4-1 Demonstraţi lema 22.2.

22.4-2 Câţi biţi sunt necesari pentru a memora $\dim(x)$, pentru fiecare nod x? Dar pentru $\mathit{rang}[x]$?

22.4-3 Folosind lema 22.2 şi corolarul 22.5, daţi o demonstraţie simplă pentru timpul de execuţie $O(m \lg n)$ pentru operaţiile pe o pădure de mulţimi disjuncte cu reuniune după rang, dar fără comprimarea drumului.

22.4-4 ⋆ Să presupunem că modificăm regula de atribuire a ponderilor astfel încât să atribuim o pondere bloc pentru ultimul nod din drumul de căutare al cărui rang este în blocul j pentru $j = 0, 1, ..., \lg^* n - 1$. Altfel atribuim nodului o pondere drum. Dacă un nod este un fiu al rădăcinii şi nu este ultimul nod al unui bloc, îi este asociată o pondere drum şi nu o pondere bloc. Arătaţi că pot fi atribuite $\Omega(m)$ ponderi drum unui nod dat cât timp rangul său este în blocul dat j.

Probleme

22-1 Minim off-line

Problema ***minimului off-line*** impune menţinerea unei mulţimi dinamice T de elemente din domeniul $\{1, 2, \ldots, n\}$, asupra căreia se pot face operaţii INSEREAZĂ şi EXTRAGE-MIN. Se consideră o secvenţă S de n apeluri INSEREAZĂ şi m apeluri EXTRAGE-MIN, prin care fiecare cheie din mulţimea $\{1, 2, \ldots, n\}$ este inserată exact o dată. Dorim să determinăm ce cheie este returnată la fiecare apel al operaţiei EXTRAGE-MIN. Mai exact, dorim să completăm un tablou *extrase*$[1..m]$, unde *extrase*$[i]$ este cheia returnată de al i-lea apel EXTRAGE-MIN, pentru $i = 1, 2, \ldots, m$. Problema este "off-line" în sensul că ne este permisă procesarea întregii secvenţe S înainte de determinarea valorii vreunei chei returnate.

a. În următorul exemplu al problemei de minim off-line, fiecare operaţie INSEREAZĂ este reprezentată de un număr şi fiecare operaţie EXTRAGE-MIN este reprezentată de litera E:

$$4, 8, \mathrm{E}, 3, \mathrm{E}, 9, 2, 6, \mathrm{E}, \mathrm{E}, \mathrm{E}, 1, 7, \mathrm{E}, 5.$$

Completaţi tabloul *extrase* cu valorile corecte.

Pentru a dezvolta un algoritm pentru această problemă, împărţim secvenţa S în subsecvenţe omogene. Astfel reprezentăm S prin

$$\mathrm{I}_1, \mathrm{E}, \mathrm{I}_2, \mathrm{E}, \mathrm{I}_3, \ldots, \mathrm{I}_m, \mathrm{E}, \mathrm{I}_{m+1},$$

unde fiecare E reprezintă un singur apel EXTRAGE-MIN şi fiecare I_j reprezintă o secvenţă (posibil vidă) de apeluri INSEREAZĂ. La început, pentru fiecare subsecvenţă I_j, formăm o mulţime K_j care conţine cheile inserate prin operaţiile corespunzătoare; mulţimea K_j este vidă dacă I_j este vidă. Apoi executăm următorul algoritm.

MINIM-OFF-LINE(m, n)
1: **pentru** $i \leftarrow 1, n$ **execută**
2: determină j astfel încât $i \in K_j$
3: **dacă** $j \neq m + 1$ **atunci**
4: *extrase*$[j] \leftarrow i$
5: fie l cea mai mică valoare mai mare decât j pentru care mulţimea K_l există
6: $K_l \leftarrow K_j \cup K_l$, distrugând K_j
7: **returnează** *extrase*

b. Argumentaţi corectitudinea tabloului *extrase*, returnat de algoritmul MINIM-OFF-LINE.

c. Descrieţi cum poate fi folosită structura de date mulţimi disjuncte, pentru a implementa eficient MINIM-OFF-LINE. Daţi o margine cât mai precisă pentru timpul de execuţie, în cazul cel mai defavorabil, al implementării voastre.

22-2 Determinarea adâncimii

În ***problema determinării adâncimii*** reţinem o pădure $\mathcal{F} = \{T_j\}$ de arbori cu rădăcină asupra căreia se execută trei operaţii:

FORMEAZĂ-ARBORE(v) creează un arbore al cărui singur nod este v.

GĂSEŞTE-ADÂNCIME(v) returnează adâncimea nodului v în arborele care îl conţine.

ADAUGĂ-SUBARBORE(r, v) face ca nodul r, ce este presupus a fi rădăcina unui arbore, să devină fiu al nodului v, care aparţine altui arbore decât r, dar nu este obligatoriu rădăcina acelui arbore.

a. Presupunem că folosim o reprezentare a arborilor similară cu cea a pădurilor de mulţimi disjuncte: $p[v]$ este părintele nodului v, excepţie făcând rădăcina pentru care $p[v] = v$. Dacă implementăm ADAUGĂ-SUBARBORE(r, v) prin setarea $p[r] \leftarrow v$ şi GĂSEŞTE-ADÂNCIME(v) prin parcurgerea drumului de căutare în sus spre rădăcină, returnând numărul tuturor nodurilor întâlnite, diferite de v, arătaţi că timpul de execuţie, în cazul cel mai defavorabil, al secvenţei de m operaţii FORMEAZĂ-ARBORE, GĂSEŞTE-ADÂNCIME şi ADAUGĂ-SUBARBORE este $\Theta(m^2)$.

Putem reduce timpul de execuţie, în cazul cel mai defavorabil, folosind euristicile reuniunii după rang şi comprimării drumului. Folosim o pădure de mulţimi disjuncte $\mathcal{S} = \{\mathcal{S}_i\}$, unde fiecare mulţime S_i (care este un arbore) corespunde unui arbore T_i din pădurea $\mathcal{F}$. Totuşi, structura arborelui din mulţimea S_i nu corespunde în mod obligatoriu structurii arborelui T_i. De fapt, implementarea lui S_i nu înregistrează exact relaţiile părinte-fiu, dar ne permite să determinăm adâncimea oricărui nod din T_i.

Ideea de bază este de a menţine în fiecare nod v o "pseudodistanţă" $d[v]$, care este definită astfel încât suma pseudodistanţelor de-a lungul drumului de la v la rădăcina mulţimii S_i din care face parte v, să fie egală cu adâncimea lui v în T_i. Aceasta înseamnă că, dacă drumul de la v la rădăcina sa din S_i este $v_0, v_1, \ldots, v_k$, unde $v_0 = v$ şi v_k este rădăcina lui S_i, atunci adâncimea lui v în T_i este $\sum_{j=0}^{k} d[v_j]$.

b. Daţi o implementare pentru FORMEAZĂ-ARBORE.

c. Arătaţi cum trebuie modificată operaţia GĂSEŞTE-MULŢIME pentru a implementa procedura GĂSEŞTE-ADÂNCIME. Implementarea trebuie să realizeze comprimarea drumului şi timpul de execuţie trebuie să fie liniar, relativ la lungimea drumului de căutare. Asiguraţi-vă că implementarea actualizează corect pseudodistanţele.

d. Arătaţi cum trebuie modificate procedurile REUNEŞTE şi UNEŞTE pentru a implementa ADAUGĂ-SUBARBORE(r, v), ce combină mulţimile care conţin pe r şi pe v. Asiguraţi-vă că implementarea actualizează corect pseudodistanţele. Observaţi că rădăcina unei mulţimi S_i nu este în mod necesar rădăcina arborelui T_i corespunzător.

e. Daţi o margine cât mai precisă pentru timpul de execuţie, în cazul cel mai defavorabil, pentru o secvenţă de m operaţii FORMEAZĂ-ARBORE, GĂSEŞTE-ADÂNCIME şi ADAUGĂ-SUBARBORE, dintre care n sunt operaţii FORMEAZĂ-ARBORE.

22-3 Algoritmul off-line al lui Tarjan pentru calculul celui mai apropiat strămoş

Cel mai apropiat strămoş a două noduri u şi v, într-un arbore cu rădăcină T este nodul w care este strămoş al ambelor noduri u şi v şi are cea mai mare adâncime în T. În ***problema off-line de determinare a celui mai apropiat strămoş***, se dă un arbore T cu rădăcină şi

o mulţime arbitrară $P = \{\{u, v\}\}$ de perechi neordonate de noduri din T; dorim să determinăm cel mai apropiat strămoş al fiecărei perechi din P.

Pentru a rezolva această problemă, procedura următoare realizează o parcurgere a arborelui T la apelul iniţial SCMA(*rădăcină*$[T]$). Presupunem că fiecare nod este colorat ALB înainte de parcurgere.

```
SCMA(u)
 1: FORMEAZĂ-MULŢIME(u)
 2: strămoş[GĂSEŞTE-MULŢIME (u)] ← u
 3: pentru fiecare fiu v al lui u din T execută
 4:    SCMA(v)
 5:    REUNEŞTE(u, v)
 6:    strămoş[GĂSEŞTE-MULŢIME(u)] ← u
 7: culoare[u] ← NEGRU
 8: pentru fiecare nod v pentru care {u, v} ∈ P execută
 9:    dacă culoare[v]= NEGRU atunci
10:       afişează "cel mai apropiat strămoş pentru" u "şi" v "este" strămoş[GĂSEŞTE-MULŢIME(v)]
```

a. Argumentaţi de ce linia 10 este executată exact o dată pentru fiecare pereche $\{u, v\} \in P$.

b. Demonstraţi că la momentul apelului SCMA(u), numărul de mulţimi din structura de date mulţimi disjuncte este egal cu adâncimea lui u în T.

c. Demonstraţi că procedura SCMA afişează corect strămoşul cel mai apropiat al lui u şi v, pentru fiecare pereche $\{u, v\} \in P$.

d. Analizaţi timpul de execuţie al procedurii SCMA, presupunând că se foloseşte implementarea pentru structura de date mulţimi disjuncte din secţiunea 22.3.

Note bibliografice

Majoritatea rezultatelor importante legate de structura de date mulţimi disjuncte se datorează, măcar în parte, lui R. E. Tarjan. Marginea superioară de $O(m\alpha(m, n))$ a fost dată prima dată de Tarjan [186, 188]. Marginea superioară $O(m \lg^* n)$ a fost demonstrată anterior de Hopcroft şi Ullman [4, 103]. Tarjan şi van Leeuwen [190] analizează variante ale euristicii comprimării drumului, incluzând "metode cu o singură trecere", care oferă uneori factori constanţi mai buni ca performanţă decât metodele cu două treceri. Gabow şi Tarjan [76] arată că în anumite aplicaţii, operaţiile cu mulţimi disjuncte pot fi executate într-un timp $O(m)$.

Tarjan [187] arată că este necesară o margine inferioară de timp $\Omega(m\alpha(m, n))$ pentru operaţiile pe orice structură de date mulţimi disjuncte care satisfac anumite condiţii tehnice. Această margine inferioară a fost mai târziu generalizată de către Fredman şi Saks [74], care arată că în cazul cel mai defavorabil, trebuie să fie accesate $\Omega(m\alpha(m, n))$ cuvinte de memorie de lungime $(\lg n)$ biţi.

VI Algoritmi de grafuri

Introducere

Grafurile sunt structuri de date foarte răspândite în ştiinţa calculatoarelor, iar algoritmii de grafuri sunt fundamentali în acest domeniu. Există sute de probleme de calcul definite în termeni de grafuri. În această parte, ne vom ocupa de câteva dintre cele mai semnificative.

În capitolul 23 se discută modurile de reprezentare a grafurilor pe calculator şi se tratează câţiva algoritmi bazaţi pe căutarea într-un graf, folosind căutarea în lăţime sau cea în adâncime. Sunt prezentate două aplicaţii ale căutării în adâncime: sortarea topologică a unui graf orientat aciclic şi descompunerea unui graf orientat în componentele sale tare conexe.

În capitolul 24 se descrie determinarea unui arbore de acoperire minim al unui graf. Un astfel de arbore este definit ca fiind cea mai "ieftină" cale de a conecta toate vârfurile atunci când fiecare muchie are un cost asociat. Algoritmii de determinare a arborilor minimi de acoperire sunt exemple foarte bune de algoritmi greedy (vezi capitolul 17).

În capitolele 25 şi 26 se tratează problema determinării drumurilor minime între vârfuri, când fiecare muchie are asociată o lungime sau un cost. În capitolul 25 se tratează determinarea drumurilor minime de la un nod sursă dat la toate celelalte vârfuri, iar în capitolul 26 se tratează determinarea drumurilor minime între orice pereche de vârfuri.

În final, în capitolul 27 se discută modul de calculare a unui flux maxim de material într-o reţea (graf orientat) având specificate sursa de material, destinaţia şi cantităţile de material care pot traversa fiecare muchie orientată (arc). Această problemă generală se regăseşte sub multe forme, iar un algoritm bun pentru a calcula fluxuri maxime poate fi folosit pentru rezolvarea eficientă a unei game variate de probleme înrudite.

Pentru evaluarea timpului de execuţie al unui algoritm de grafuri pe un graf dat $G = (V, E)$, de obicei, vom măsura dimensiunea intrării în funcţie de numărul de vârfuri $|V|$ şi al numărului de muchii $|E|$ al grafului. Rezultă că există doi parametri relevanţi care descriu dimensiunea intrării, şi nu unul singur. Vom adopta o convenţie de notaţie răspândită pentru aceşti parametri. În cadrul notaţiei asimptotice (de exemplu, notaţia O sau Θ şi *numai* în cadrul acesteia) simbolul V înseamnă $|V|$, iar simbolul E înseamnă $|E|$. De exemplu, putem scrie "algoritmul are ordinul de complexitate $O(VE)$", aceasta însemnând, de fapt, "algoritmul are ordinul de complexitate $O(|V||E|)$". Această notaţie permite obţinerea unor formule mai lizibile pentru ordinul de complexitate al algoritmilor, fără riscul unor ambiguităţi.

O altă convenţie pe care o vom adopta apare în pseudocod. Vom nota mulţimea vârfurilor unui graf G cu $V[G]$ şi mulţimea muchiilor sale cu $E[G]$. Cu alte cuvinte, din punctul de vedere al pseudocodului, mulţimea vârfurilor şi cea a muchiilor sunt atribute ale unui graf.

23 Algoritmi elementari de grafuri

Acest capitol prezintă câteva metode de reprezentare a unui graf precum şi unele metode de căutare într-un graf. Căutarea într-un graf înseamnă parcurgerea, în mod sistematic, a muchiilor grafului astfel încât să parcurgem vârfurile grafului. Un algoritm de căutare într-un graf poate descoperi multe informaţii despre structura grafului respectiv. Mulţi algoritmi de grafuri încep prin a căuta în graful de intrare pentru a obţine această informaţie structurală. Alţi algoritmi de grafuri sunt simple rafinări ale unor algoritmi de căutare de bază.

În secţiunea 23.1 se discută cele mai uzuale două moduri de reprezentare a grafurilor, ca structuri de date: liste de adiacenţă şi matrice de adiacenţă. În secţiunea 23.2 este prezentat un algoritm simplu de căutare în grafuri numit căutarea în lăţime şi se prezintă modul de creare al arborelui de lăţime. În secţiunea 23.3, se prezintă căutarea în adâncime şi se demonstrează câteva rezultate consacrate despre ordinea în care căutarea în adâncime vizitează vârfurile. În secţiunea 23.4, se prezintă prima aplicaţie a căutării în adâncime: sortarea topologică a unui graf orientat aciclic. O a doua aplicaţie a căutării în adâncime, găsirea componentelor tare conexe ale unui graf orientat, este prezentată în secţiunea 23.5.

23.1. Reprezentările grafurilor

Există două moduri standard de reprezentare a unui graf $G = (V, E)$: ca o mulţime de liste de adiacenţă sau ca o matrice de adiacenţă. Reprezentarea prin liste de adiacenţă este, de obicei, preferată, pentru că oferă un mod compact de reprezentare a grafurilor ***rare*** – acelea pentru care $|E|$ este mult mai mic decât $|V|^2$. Majoritatea algoritmilor de grafuri prezentaţi în această carte presupun că un graf de intrare este reprezentat în forma cu liste de adiacenţă. Totuşi, o reprezentare cu matrice de adiacenţă poate fi preferată atunci când graful este ***dens*** – $|E|$ este aproximativ egal cu $|V|^2$ – sau atunci când trebuie să decidem, rapid, dacă există o muchie ce conectează două vârfuri date. De exemplu, doi dintre algoritmii de drum minim între toate perechile de vârfuri, presupun că grafurile lor de intrare sunt reprezentate prin matrice de adiacenţă.

Reprezentarea prin liste de adiacenţă a unui graf $G = (V, E)$ constă într-un tablou Adj cu $|V|$ liste, una pentru fiecare vârf din V. Pentru fiecare $u \in V$, lista de adiacenţă $Adj[u]$ conţine (pointeri la) toate vârfurile v pentru care există o muchie $(u, v) \in E$. Cu alte cuvinte, $Adj[u]$ este formată din totalitatea vârfurilor adiacente lui u în G. De obicei, vârfurile din fiecare listă de adiacenţă sunt memorate într-o ordine arbitrară. Figura 23.1(b) ilustrează o reprezentare prin liste de adiacenţă a grafului neorientat din figura 23.1(a). În mod similar, figura 23.2(b) ilustrează o reprezentare prin liste de adiacenţă a grafului orientat din figura 23.2(a).

Dacă G este un graf orientat, suma lungimilor tuturor listelor de adiacenţă este $|E|$, deoarece un arc de forma (u, v) este reprezentat prin apariţia lui v în $Adj[u]$. Dacă G este un graf neorientat, suma lungimilor tuturor listelor de adiacenţă este $2|E|$, deoarece, dacă (u, v) este o muchie, atunci u apare în lista de adiacenţă a lui v şi invers. Indiferent dacă un graf este orientat sau nu, reprezentarea prin liste de adiacenţă are proprietatea dezirabilă că dimensiunea memoriei necesare este $O(\max(V, E)) = O(V + E)$.

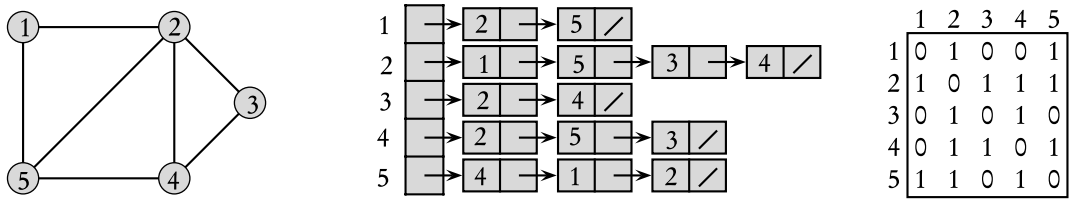


Figura 23.1 Două reprezentări ale unui graf neorientat. **(a)** Un graf neorientat G având cinci vârfuri şi şapte muchii. **(b)** O reprezentare prin liste de adiacenţă a lui G. **(c)** Reprezentarea prin matrice de adiacenţă a lui G.

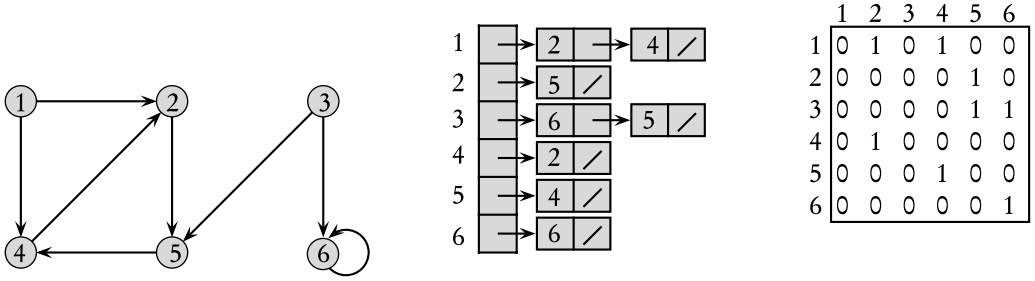


Figura 23.2 Două reprezentări ale unui graf orientat. **(a)** Un graf orientat G având şase vârfuri şi opt muchii. **(b)** O reprezentare prin liste de adiacenţă a lui G. **(c)** Reprezentarea prin matrice de adiacenţă a lui G.

Listele de adiacenţă pot fi uşor adaptate în scopul reprezentării ***grafurilor cu costuri***,[1] adică acele grafuri în care fiecărei muchii i s-a asociat un ***cost*** dat, de obicei, de o ***funcţie de cost*** $w : E \to \mathbb{R}$. De exemplu, fie $G = (V, E)$ un graf cu costuri având funcţia de cost w. Costul $w(u, v)$ al muchiei $(u, v) \in E$ este memorat pur şi simplu împreună cu vârful v în lista de adiacenţă a lui u. Reprezentarea prin liste de adiacenţă este foarte robustă în sensul că poate fi modificată pentru a suporta multe alte variante de grafuri.

Un dezavantaj al reprezentării prin liste de adiacenţă este acela că nu există altă cale mai rapidă pentru a determina dacă o muchie dată (u, v) aparţine grafului, decât căutarea lui v în lista de adiacenţă $Adj[u]$. Acest dezavantaj poate fi remediat folosind o reprezentare a grafului prin matrice de adiacenţă, dar folosind asimptotic mai multă memorie.

Pentru ***reprezentarea prin matrice de adiacenţă*** a unui graf $G = (V, E)$, presupunem că vârfurile sunt numerotate cu $1, 2, ..., |V|$ într-un mod arbitrar. Reprezentarea prin matrice de adiacenţă a grafului G constă într-o matrice $A = (a_{ij})$ de dimensiuni $|V| \times |V|$ astfel încât:

$$a_{ij} = \begin{cases} 1 & \text{dacă } (i, j) \in E, \\ 0 & \text{altfel.} \end{cases}$$

În figurile 23.1(c) şi 23.2(c) sunt prezentate matricele de adiacenţă ale grafurilor neorientate şi orientate din figurile 23.1(a) şi, respectiv, 23.2(a). Necesarul de memorie pentru matricea de

[1] În literatura de specialitate în limba română se foloseşte uneori şi termenul de "graf ponderat" – n. t.

adiacenţă a unui graf este $\Theta(V^2)$ şi nu depinde de numărul de muchii ale grafului.

Se poate observa simetria faţă de diagonala principală a matricei de adiacenţă din figura 23.1(c). Se defineşte ***transpusa*** unei matrice $A = (a_{ij})$ ca fiind matricea $A^T = (a_{ij}^T)$, dată de $a_{ij}^T = a_{ji}$. Deoarece într-un graf neorientat (u, v) şi (v, u) reprezintă aceeaşi muchie, matricea de adiacenţă A a unui graf neorientat este propria sa transpusă: $A = A^T$. În anumite aplicaţii, este avantajos să stocăm numai elementele situate pe şi deasupra diagonalei principale, reducând astfel memoria necesară pentru stocarea grafului cu aproape 50 de procente.

La fel ca reprezentarea prin liste de adiacenţă, reprezentarea prin matrice de adiacenţă poate fi folosită şi pentru grafuri cu cost. De exemplu, dacă $G = (V, E)$ este un graf cu cost, având costul unei muchii dat de funcţia w, costul $w(u, v)$ al muchiei $(u, v) \in E$ este memorat pur şi simplu ca un element din linia u şi coloana v a matricei de adiacenţă. Dacă o muchie nu există, elementul corespunzător în matrice poate fi NIL, deşi pentru multe probleme este convenabilă folosirea unei valori ca 0 sau ∞.

Deşi reprezentarea prin liste de adiacenţă este asimptotic cel puţin la fel de eficientă ca reprezentarea prin matrice de adiacenţă, simplitatea matricei de adiacenţă o poate face preferabilă, atunci când grafurile au un număr relativ mic de vârfuri. Mai mult, dacă graful este fără costuri, există un avantaj suplimentar de stocare pentru reprezentarea prin matrice de adiacenţă. În locul folosirii unui cuvânt de memorie pentru fiecare element din matrice, matricea de adiacenţă foloseşte doar un bit pentru fiecare element.

Exerciţii

23.1-1 Dată fiind o reprezentare prin liste de adiacenţă a unui graf orientat, cât timp durează calcularea gradului exterior al fiecărui vârf? Cât timp durează calcularea gradelor interioare?

23.1-2 Daţi o reprezentare prin liste de adiacenţă pentru un arbore binar complet cu 7 vârfuri. Daţi o reprezentare prin matrice de adiacenţă echivalentă. Presupuneţi că vârfurile sunt numerotate de la 1 la 7 ca într-o mulţime binară.

23.1-3 ***Transpusul*** unui graf orientat $G = (V, E)$ este graful $G^T = (V, E^T)$, unde $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Astfel, G^T este G cu toate muchiile sale inversate. Descrieţi câţiva algoritmi eficienţi pentru calcularea lui G^T din G, pentru reprezentarea prin liste de adiacenţă şi prin matrice de adiacenţă a lui G. Analizaţi timpii de execuţie ai algoritmilor.

23.1-4 Dată fiind o reprezentare prin liste de adiacenţă a unui multigraf $G = (V, E)$, descrieţi un algoritm care să ruleze într-un timp $O(V + E)$, care să calculeze reprezentarea prin liste de adiacenţă a grafului neorientat "echivalent" $G' = (V, E')$, unde în E' toate muchiile multiple între două vârfuri din E sunt înlocuite printr-o singură muchie şi toate ciclurile de la un nod la el însuşi sunt anulate.

23.1-5 ***Pătratul*** unui graf orientat $G = (V, E)$ este graful $G^2 = (V, E^2)$ în care $(u, w) \in E^2$ dacă şi numai dacă pentru un vârf oarecare $v \in V$, atât (u, v), cât şi (v, w) aparţin lui E. Aceasta înseamnă că G^2 conţine o muchie între u şi w dacă G conţine un drum cu exact două muchii între u şi w. Descrieţi câţiva algoritmi eficienţi pentru a calcula G^2 din G, folosind reprezentarea prin liste de adiacenţă respectiv reprezentarea prin matrice de adiacenţă a lui G. Analizaţi timpii de execuţie ai algoritmilor propuşi.

23.1-6 Când se foloseşte o reprezentare prin matrice de adiacenţă, cei mai mulţi algoritmi de grafuri necesită timpul $\Theta(V^2)$, dar există şi excepţii. Arătaţi că, pentru a determina dacă un graf orientat are sau nu o ***destinaţie*** – un vârf cu numărul muchiilor ce intră în el egal cu $|V| - 1$ şi numărul muchiilor ce ies din el egal cu 0 – există un algoritm de timp $O(V)$, chiar dacă se foloseşte o reprezentare prin matrice de adiacenţă.

23.1-7 Matricea de incidenţă a unui graf orientat $G = (V, E)$ este o matrice $B = (b_{ij})$ de dimensiuni $|V| \times |E|$ astfel încât

$$b_{ij} = \begin{cases} -1 & \text{dacă muchia } j \text{ pleacă din vârful } i, \\ 1 & \text{dacă muchia } j \text{ intră în vârful } i, \\ 0 & \text{altfel.} \end{cases}$$

Arătaţi ce reprezintă elementele produsului matriceal BB^T, unde B^T este transpusa lui B.

23.2. Căutarea în lăţime

Căutarea în lăţime este unul din cei mai simpli algoritmi de căutare într-un graf şi arhetipul pentru mulţi algoritmi de grafuri importanţi. Algoritmul lui Dijkstra pentru determinarea drumurilor minime de la un nod sursă la toate celelalte (capitolul 25) şi algoritmul lui Prim pentru determinarea arborelui parţial de cost minim (secţiunea 24.2) folosesc idei similare celor din algoritmul de căutare în lăţime.

Dat fiind un graf $G = (V, E)$ şi un nod ***sursă*** s, căutarea în lăţime explorează sistematic muchiile lui G pentru a "descoperi" fiecare nod care este accesibil din s. Algoritmul calculează distanţa (cel mai mic număr de muchii) de la s la toate aceste vârfuri accesibile. El produce un "arbore de lăţime" cu rădăcina s, care conţine toate aceste vârfuri accesibile. Pentru fiecare vârf v accesibil din s, calea din arborele de lăţime de la s la v corespunde unui "cel mai scurt drum" de la s la v în G, adică un drum care conţie un număr minim de muchii. Algoritmul funcţionează atât pe grafuri orientate cât şi pe grafuri neorientate.

Căutarea în lăţime este numită astfel pentru că lărgeşte, uniform, frontiera dintre nodurile descoperite şi cele nedescoperite, pe lăţimea frontierei. Aceasta înseamnă că algoritmul descoperă toate vârfurile aflate la distanţa k faţă de s înainte de a descoperi vreun vârf la distanţa $k + 1$.

Pentru a ţine evidenţa avansării, căutarea în lăţime colorează fiecare nod cu alb, gri sau negru. Toate vârfurile sunt colorate la început cu alb şi pot deveni apoi gri sau negre. Un vârf este ***descoperit*** când este întâlnit prima dată în timpul unei căutări, moment în care nu mai este alb. De aceea, vârfurile gri şi negre sunt descoperite, dar algoritmul face diferenţa între ele pentru a fi sigur că procesul de căutare are loc pe lăţime. Dacă $(u, v) \in E$ şi vârful u este negru, atunci vârful v este sau gri sau negru, adică toate vârfurile adiacente unui nod negru au fost descoperite. Vârfurile gri pot avea vârfuri adiacente albe. Acestea reprezintă frontiera dintre vârfurile descoperite şi cele nedescoperite.

Algoritmul de căutare în lăţime construieşte un arbore de lăţime ce conţine, iniţial, numai rădăcina sa, care este vârful s. Oricând un vârf alb v este descoperit în cursul parcurgerii listei de adiacenţă a unui vârf u deja descoperit, vârful v şi muchia (u, v) sunt adăugate în arbore. În acest caz, spunem că u este ***predecesorul*** sau ***părintele*** lui v în arborele de lăţime. Deoarece un vârf este descoperit cel mult o dată, el poate avea cel mult un părinte. Relaţiile de strămoş

şi descendent, din arborele de lăţime, sunt definite relativ la rădăcina s în mod obişnuit: dacă u se află pe un drum din arbore de la rădăcina s la vârful v, atunci u este un strămoş al lui v, iar v este un descendent al lui u.

Procedura de căutare în lăţime CL, de mai jos, presupune că graful de intrare $G = (V, E)$ este reprezentat folosind liste de adiacenţă. Algoritmul întreţine mai multe structuri de date adiţionale pentru fiecare vârf din graf. Culoarea fiecărui vârf $u \in V$ este memorată în variabila $color[u]$, şi predecesorul lui u este memorat în variabila $\pi[u]$. Dacă u nu are nici un predecesor (de exemplu, dacă $u = s$ sau u nu a fost descoperit), atunci $\pi[u] =$ NIL. Distanţa de la sursa s la vârful u, calculată de algoritm, este memorată în $d[u]$. De asemenea, algoritmul foloseşte o coadă Q de tipul primul sosit, primul servit (a se vedea secţiunea 11.1) pentru a prelucra mulţimea de vârfuri gri.

```
CL(G, s)
 1: pentru fiecare vârf u ∈ V[G] − {s} execută
 2:    color[u] ← ALB
 3:    d[u] ← ∞
 4:    π[u] ← NIL
 5: color[s] ← GRI
 6: d[s] ← 0
 7: π[s] ← NIL
 8: Q ← {s}
 9: cât timp Q ≠ ∅ execută
10:    u ← cap[Q]
11:    pentru fiecare vârf v ∈ Adj[u] execută
12:       dacă color[v] = ALB atunci
13:          color[v] ← GRI
14:          d[v] ← d[u] + 1
15:          π[v] ← u
16:          PUNE-ÎN-COADĂ(Q, v)
17:    SCOATE-DIN-COADĂ(Q)
18:    color[u] ← NEGRU
```

Figura 23.3 ilustrează efectele algoritmului CL pe un graf de test.

Procedura CL lucrează în felul următor: Liniile 1–4 colorează fiecare vârf în alb, atribuie lui $d[u]$ valoarea ∞ pentru fiecare vârf u şi setează părintele fiecărui vârf pe NIL. Linia 5 colorează vârful sursă s cu gri, deoarece el este considerat descoperit atunci când procedura începe. Linia 6 iniţializează $d[s]$ cu 0, iar linia 7 iniţializează predecesorul sursei cu NIL. Linia 8 iniţializează Q ca fiind o coadă ce conţine un singur vârf, şi anume s. Ulterior, Q conţine mulţimea vârfurilor gri.

Bucla principală a programului este conţinută în liniile 9–18. Iteraţia are loc cât timp mai există vârfuri gri, adică vârfuri descoperite a căror listă de adiacenţă nu a fost examinată în întregime. Linia 10 determină vârful gri u din capul cozii Q. Bucla **pentru**, din liniile 11–16, consideră fiecare vârf v din lista de adiacenţă a lui u. Dacă v este alb, atunci el nu a fost încă descoperit, şi algoritmul îl descoperă executând liniile 13–16. La început, el este colorat cu gri, iar distanţa $d[v]$ este iniţializată cu $d[u] + 1$. Apoi, u este înregistrat ca părinte al său. În final, vârful v este plasat la sfârşitul cozii Q. Când toate vârfurile din lista de adiacenţă a lui u au fost examinate, u este scos din Q şi colorat cu negru în liniile 17–18.

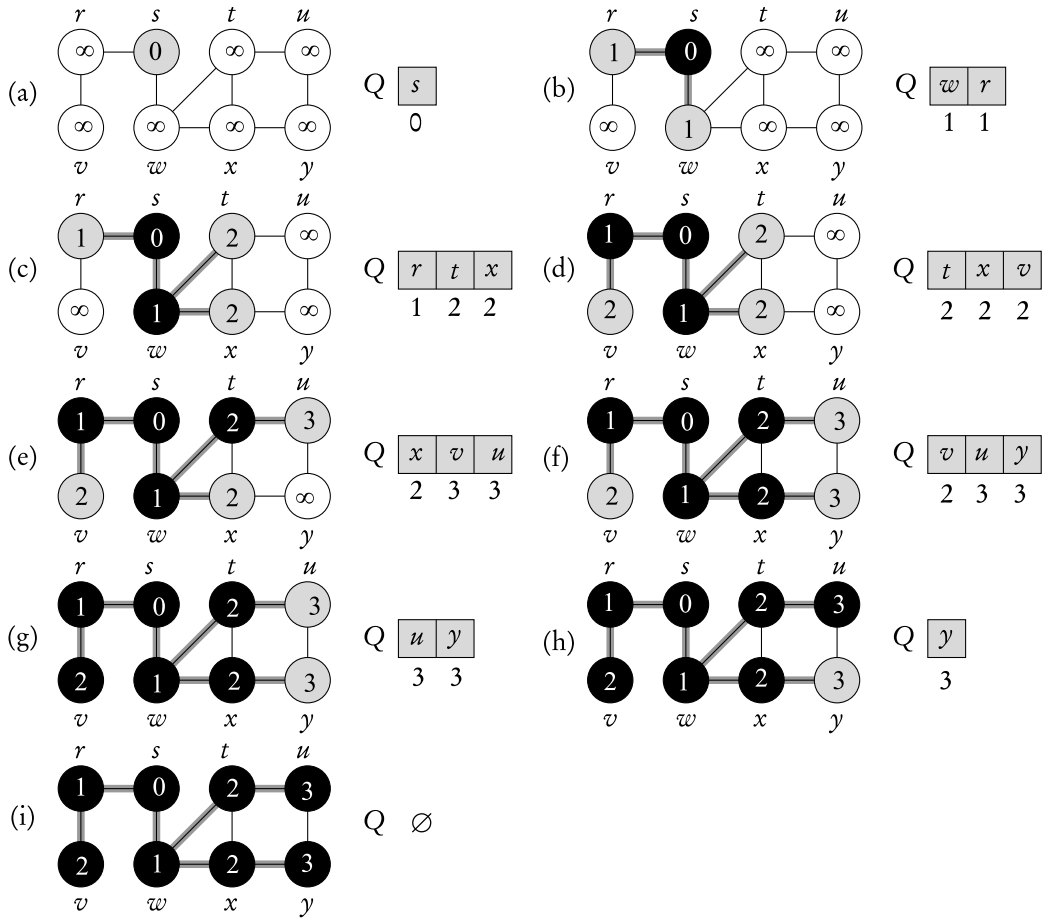


Figura 23.3 Execuţia algoritmului CL pe un graf neorientat. Muchiile arborilor sunt prezentate haşurat aşa cum sunt produse de CL. Împreună cu fiecare vârf u, este prezentată $d[u]$. Coada Q este prezentată la începutul fiecărei iteraţii a buclei **cât timp** din liniile 9–18. Distanţele vârfurilor sunt prezentate lângă vârfurile din coadă.

Analiză

Înainte de a descrie variatele proprietăţi ale căutării în lăţime, vom aborda o problemă mai uşoară, şi anume, analiza timpului de execuţie a acestui algoritm pe un graf de intrare $G = (V, E)$. După iniţializare, nici un vârf nu este făcut vreodată alb, şi, astfel, testul din linia 12 asigură faptul că fiecare vârf este pus în coadă cel mult o dată şi, de aceea, acesta va fi scos din coadă cel mult o dată. Operaţiile de punere şi de extragere a unui element din coadă se fac într-un timp $O(1)$, deci timpul total alocat operaţiilor efectuate cu coada este $O(V)$. Pentru că lista de adiacenţă a fiecărui vârf este examinată doar atunci când vârful este scos din coadă, lista de adiacenţă a fiecărui vârf este examinată cel mult o dată. Deoarece suma lungimilor tuturor listelor de adiacenţă este $\Theta(E)$, timpul cel mai defavorabil pentru parcurgerea tuturor listelor de adiacenţă este $O(E)$. Timpul pentru iniţializare este $O(V)$ şi, astfel, timpul total de execuţie a

algoritmului CL este $O(V+E)$. Astfel, căutarea în lăţime rulează într-un timp liniar, ce depinde de mărimea reprezentării prin liste de adiacenţă a grafului G.

Drumuri de lungime minimă

La începutul acestei secţiuni, am afirmat că algoritmul de căutare în lăţime determină distanţa de la un nod sursă dat, $s \in V$, la fiecare nod accesibil dintr-un graf $G = (V, E)$. Definim ***lungimea celui mai scurt drum*** $\delta(s, v)$, din s în v, ca fiind numărul minim de muchii ale oricărui drum din vârful s în vârful v, sau ∞ dacă nu există nici un drum de la s la v. Un drum de lungime $\delta(s, v)$, de la s la v, se numeşte ***drum de lungime minimă***[2] de la s la v. Înainte de a demonstra că algoritmul de căutare în lăţime calculează, de fapt, lungimile drumurilor minime, vom studia o proprietate importantă a lungimilor drumurilor minime.

Lema 23.1 Fie $G = (V, E)$ un graf orientat sau neorientat, şi fie $s \in V$ un vârf arbitrar. Atunci, pentru orice muchie $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + 1.$$

Demonstraţie. Dacă u este accesibil din s, atunci şi v este accesibil din s. În acest caz, cel mai scurt drum de la s la v nu poate fi mai lung decât cel mai scurt drum de la s la u, la care se adaugă muchia (u, v), şi astfel inegalitatea este adevărată. Dacă u nu este accesibil din s, atunci $\delta(s, u) = \infty$, şi inegalitatea este adevărată. ∎

Dorim să demonstrăm că procedura CL calculează corect $d[v] = \delta(s, v)$ pentru fiecare vârf $v \in V$. Vom demonstra întâi că $d[v]$ mărgineşte superior $\delta(s, v)$.

Lema 23.2 Fie $G = (V, E)$ un graf orientat sau neorientat şi să presupunem că CL este rulat pe G pentru un vârf sursă $s \in V$ dat. Atunci, la terminare, pentru fiecare vârf $v \in V$, valoarea $d[v]$ calculată de CL satisface inegalitatea $d[v] \geq \delta(s, v)$.

Demonstraţie. Vom folosi inducţia după numărul de plasări ale unui vârf în coada Q. Ipoteza inductivă este că $d[v] \geq \delta(s, v)$ pentru orice $v \in V$. Baza acestei inducţii este situaţia imediat următoare momentului când, în linia 8 din CL, s este plasat în Q. Ipoteza inductivă este adevărată, pentru că $d[s] = 0 = \delta(s, s)$ şi $d[v] = \infty \geq \delta(s, v)$, pentru orice $v \in V - \{s\}$.

Pentru pasul inductiv, vom considera un vârf alb v care este descoperit în timpul parcurgerii listei vecinilor unui vârf u. Ipoteza inductivă implică faptul că $d[u] \geq \delta(s, u)$. Din atribuirea efectuată în linia 14 şi din lema 23.1, obţinem:

$$d[v] = d[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v).$$

Apoi, vârful v este inserat în coada Q şi nu va mai fi introdus niciodată pentru că este colorat gri şi clauza **atunci** din liniile 13–16 este executată numai pentru vârfurile albe. Astfel valoarea $d[v]$ nu se mai schimbă niciodată şi ipoteza inductivă este verificată. ∎

Pentru a demonstra că $d[v] = \delta(s, v)$, trebuie, mai întâi, să studiem mai exact cum lucrează coada Q în timpul execuţiei CL. Următoarea lemă arată că, în orice moment, există în coadă cel mult două valori distincte d.

[2] În capitolele 25 şi 26 vom generaliza studiul nostru referitor la drumurile de lungime minimă la grafurile cu costuri, în care fiecare muchie are un cost exprimat printr-o valoare reală, iar costul unui drum este dat de suma costurilor muchiilor constituente. Grafurile luate în considerare în prezentul capitol sunt fără costuri.

Lema 23.3 Să presupunem că în timpul execuţiei lui CL pe un graf $G = (V, E)$, coada Q conţine vârfurile $\langle v_1, v_2, ..., v_r \rangle$, unde v_1 este primul element al lui Q şi v_r este ultimul. Atunci $d[v_r] \leq d[v_1] + 1$ şi $d[v_i] \leq d[v_{i+1}]$ pentru $i = 1, 2, ..., r - 1$.

Demonstraţie. Demonstraţia se face prin inducţie după numărul de operaţii cu coada. Iniţial, când coada îl conţine doar pe s, lema este evident adevărată.

Pentru pasul inductiv, trebuie să demonstrăm că lema este adevărată după ce am scos un vârf din listă, respectiv, am pus unul. Dacă vârful v_1 din capul listei este scos din listă, noul cap al listei este v_2. (În cazul în care coada devine vidă, lema este evident adevărată.) Dar atunci, avem $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, iar inegalităţile rămase sunt neafectate. Astfel, lema continuă cu v_2 pe post de cap al listei. Punerea unui vârf în listă necesită o examinare mai atentă a codului. În linia 16 a CL, când vârful v este pus în listă, devenind astfel v_{r+1}, capul v_1 al lui Q este, de fapt, vârful u a cărui listă de adiacenţă este examinată în acel moment. Astfel $d[v_{r+1}] = d[v] = d[u] + 1 = d[v_1] + 1$. Avem, de asemenea, $d[v_r] \leq d[v_1] + 1 = d[u] + 1 = d[v] = d[v_{r+1}]$ şi inegalităţile rămase sunt neafectate. Astfel lema este adevărată când v este pus în coadă. ∎

Putem, acum, demonstra că algoritmul de căutare în lăţime calculează corect lungimile celor mai scurte drumuri.

Teorema 23.4 (Corectitudinea căutării în lăţime) Fie $G = (V, E)$ un graf orientat sau neorientat şi să presupunem că CL este rulat pe G pentru un vârf sursă $s \in V$ dat. Atunci, în timpul execuţiei sale, CL descoperă fiecare vârf $v \in V$ care este accesibil din nodul sursă s şi, după terminare, $d[v] = \delta(s, v)$ pentru orice $v \in V$. Mai mult, pentru orice vârf $v \neq s$, accesibil din s, unul din cele mai scurte drumuri de la s la v este cel mai scurt drum de la s la $\pi[v]$ urmat de muchia $(\pi[v], v)$.

Demonstraţie. Începem cu cazul în care v nu este accesibil din s. Deoarece lema 23.2 afirmă că $d[v] \geq \delta(s, v) = \infty$, valorii $d[v]$ a vârfului v nu i se poate atribui niciodată o valoare finită în linia 14. Prin inducţie, nu poate exista un prim vârf a cărui câmp d să primească valoarea ∞ în linia 14. De aceea, linia 14 este executată doar pentru vârfurile cu valori d finite. Astfel, dacă v nu este accesibil, nu este niciodată descoperit.

Partea principală a demonstraţiei analizează cazul vârfurilor accesibile din s. Fie V_k mulţimea vârfurilor aflate la distanţa k de s, adică $V_k = \{v \in V : \delta(s, v) = k\}$. Demonstraţia începe cu o inducţie după k. Presupunem, ca ipoteză inductivă, că, pentru fiecare vârf $v \in V_k$, există exact un moment în timpul execuţiei lui CL în care

- v este gri,
- lui $d[v]$ i se atribuie valoarea k,
- dacă $v \neq s$ atunci lui $\pi[v]$ i se atribuie valoarea u, pentru un $u \in V_{k-1}$ arbitrar şi
- v este inserat în coada Q.

După cum am observat mai devreme, există cu siguranţă cel mult un astfel de moment.

Baza inducţiei este cazul $k = 0$. Deoarece s este singurul vârf la distanţa 0 de s, avem $V_0 = \{s\}$. În timpul iniţializării, s este colorat cu gri, $d[s]$ primeşte valoarea 0, iar s este plasat în Q, deci ipoteza inductivă este adevărată.

Pentru pasul inductiv, observăm că, evident, coada Q nu este niciodată vidă până când algoritmul nu se termină şi că, o dată ce un vârf u este inserat în coadă, nici $d[u]$ nici $\pi[u]$ nu

se schimbă vreodată. De aceea, din lema 23.3, dacă vârfurile sunt inserate în coadă în timpul execuţiei algoritmului în ordinea $v_1, v_2, ..., v_r$, atunci şirul de distanţe este crescător: $d[v_i] \leq d[v_{i+1}]$ pentru $i = 1, 2, ..., r-1$.

Acum, să considerăm un vârf arbitrar $v \in V_k$, unde $k \geq 1$. Proprietatea de monotonie, combinată cu faptul că $d[v] \geq k$ (din lema 23.2) şi cu ipoteza inductivă, implică faptul că v trebuie descoperit, dacă este descoperit vreodată, după ce toate vârfurile din V_{k-1} sunt puse în coadă.

Deoarece $\delta(s, v) = k$, există un drum de k muchii de la s la v, şi, astfel, există un vârf $u \in V_{k-1}$ aşa încât $(u, v) \in E$. Fără a reduce generalitatea, fie u primul astfel de vârf colorat în gri, care trebuie să existe deoarece, prin inducţie, toate vârfurile din V_{k-1} sunt gri. Codul din CL pune în coadă toate vârfurile gri şi de aici u trebuie să apară în cele din urmă în postura de cap al cozii Q în linia 10. Când u apare în postura de cap al listei, lista sa de adiacenţă este examinată şi v este descoperit. (Vârful v nu ar fi putut fi descoperit mai devreme deoarece nu este adiacent vreunui vârf din V_j pentru $j < k-1$ – altfel, v nu ar putea aparţine lui V_k – şi, prin presupunerea anterioară, u este primul vârf descoperit din V_{k-1} căruia v îi este adiacent. În linia 13 v este colorat în gri, în linia 14 se face atribuirea $d[v] = d[u] + 1 = k$, în linia 15 lui $\pi[v]$ îi este atribuită valoarea u, iar în linia 16 v este inserat în coadă. Deoarece v este un vârf arbitrar din V_k, ipoteza inductivă este astfel demonstrată.

Pentru a termina demonstraţia teoremei, să observăm că dacă $v \in V_k$ atunci pe baza celor deja demonstrate, $\pi[v] \in V_{k-1}$. Astfel putem obţine un drum de lungime minimă de la s la v, luând un drum de lungime minimă de la s la $\pi[v]$ şi traversând apoi muchia $(\pi[v], v)$. ■

Arbori de lăţime

Procedura CL construieşte un arbore de lăţime pe măsură ce caută în graf, aşa cum este ilustrat în figura 23.3. Arborele este reprezentat de câmpul π în fiecare vârf. Mai formal, pentru un graf $G = (V, E)$ cu sursa s, definim ***subgraful predecesor*** al lui G ca fiind $G_\pi = (V_\pi, E_\pi)$, unde

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

şi

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}.$$

Subgraful predecesor G_π este un ***arbore de lăţime*** dacă mulţimea V_π este formată din vârfurile accesibile din s şi, pentru orice $v \in V_\pi$, există în G_π un unic drum simplu de la s la v, care este, de asemenea, un drum de lungime minimă în G de la s la v. Un arbore de lăţime este de fapt un arbore, deoarece este conex şi $|E_\pi| = |V_\pi| - 1$ (vezi teorema 5.2). Muchiile din E_π se numesc ***muchii de arbore***.

După ce CL a fost rulat pe un graf G, pentru un vârf sursă s, următoarea lemă arată faptul că subgraful predecesor este un arbore de lăţime.

Lema 23.5 Când este rulată pe un graf orientat sau neorientat $G = (V, E)$, procedura CL construieşte π astfel încât subgraful predecesor $G_\pi = (V_\pi, E_\pi)$ este un arbore de lăţime.

Demonstraţie. În linia 15 din CL se face atribuirea $\pi[v] = u$, doar dacă $(u, v) \in E$ şi $\delta(s, v) < \infty$ – adică dacă v este accesibil din s – şi astfel V_π este constituit din vârfurile din V accesibile din

s. Deoarece G_π formează un arbore, el conţine un unic drum de la s la fiecare vârf din V_π. Aplicând inductiv teorema 23.4, tragem concluzia că fiecare astfel de drum este un drum de lungime minimă. ∎

Următoarea procedură tipăreşte vârfurile de pe un drum de lungime minimă de la s la v, presupunând că algoritmul CL a fost deja rulat pentru a calcula arborele de drumuri de lungime minimă.

TIPĂREŞTE-CALE(G, s, v)
1: **dacă** $v = s$ **atunci**
2: tipăreşte s
3: **altfel dacă** $\pi[v] =$ NIL **atunci**
4: tipăreşte "nu există nici un drum de la " s " la " v
5: **altfel**
6: TIPĂREŞTE-CALE$(G, s, \pi[v])$
7: tipăreşte v

Această procedură rulează într-un timp liniar ce depinde de numărul de vârfuri din drumul care este tipărit, deoarece fiecare apel recursiv se face pentru un drum mai scurt cu un vârf.

Exerciţii

23.2-1 Precizaţi rezultatul rulării algoritmului de căutare în lăţime pe graful orientat din figura 23.2(a) folosind vârful 3 pe post de sursă.

23.2-2 Precizaţi rezultatul rulării algoritmului de căutare în lăţime pe graful orientat din figura 23.3 folosind vârful u pe post de sursă.

23.2-3 Care este timpul de execuţie al CL dacă graful său de intrare este reprezentat printr-o matrice de adiacenţă şi algoritmul este modificat să suporte această formă de intrare?

23.2-4 Argumentaţi faptul că, pentru o căutare în lăţime, valoarea $d[u]$ atribuită unui vârf u este independentă de ordinea în care sunt date vârfurile din fiecare listă de adiacenţă.

23.2-5 Daţi un exemplu de graf orientat $G = (V, E)$, un vârf sursă $s \in V$ şi o mulţime de muchii de arbore $E_\pi \subseteq E$ astfel încât, pentru fiecare vârf $v \in V$, drumul unic din E_π de la s la v este un drum de lungime minimă în G, şi, totuşi, mulţimea de muchii E_π nu poate fi obţinută rulând CL pe G, indiferent de modul în care sunt aranjate nodurile în fiecare listă de adiacenţă.

23.2-6 Elaboraţi un algoritm eficient pentru a determina dacă un graf neorientat este bipartit.

23.2-7 ⋆ ***Diametrul*** unui arbore $T = (V, E)$ este dat de

$$\max_{u,v \in V} \delta(u, v),$$

adică diametrul este cel mai lung drum dintre toate drumurile de lungime minimă din arbore. Elaboraţi un algoritm eficient pentru a determina diametrul unui arbore şi analizaţi timpul de execuţie al algoritmului propus.

23.2-8 Fie $G = (V, E)$ un graf neorientat. Daţi un algoritm care să ruleze într-un timp $O(V+E)$ pentru a determina un drum în G care să traverseze fiecare muchie din E exact o dată în fiecare direcţie. Descrieţi cum se poate găsi calea de ieşire dintr-un labirint, presupunând că vi se dă o cantitate suficientă de bănuţi.

23.3. Căutarea în adâncime

Strategia folosită de căutarea în adâncime este, după cum ne arată şi numele, de a căuta "mai adânc" în graf oricând acest lucru este posibil. În căutarea în adâncime, muchiile sunt explorate pornind din vârful v cel mai recent descoperit care mai are încă muchii neexplorate, care pleacă din el. Când toate muchiile care pleacă din v au fost explorate, căutarea "revine" pe propriile urme, pentru a explora muchiile care pleacă din vârful din care v a fost descoperit. Această operaţie continuă până când sunt descoperite toate vârfurile accesibile din vârful sursă iniţial. Întregul proces este repetat până când toate vârfurile sunt descoperite.

La fel ca în cazul căutării în lăţime, de fiecare dată când un vârf v este descoperit în timpul unei examinări a listei de adiacenţă a unui vârf u, deja descoperit, căutarea în adâncime înregistrează acest eveniment punându-l pe u în câmpul predecesorului lui v, $\pi[v]$. Spre deosebire de căutarea în lăţime, unde subgraful predecesor formează un arbore, subgraful predecesor produs de căutarea în adâncime poate fi compus din mai mulţi arbori, deoarece căutarea poate fi repetată pentru surse multiple. De aceea, ***subgraful predecesor*** al unei căutări în adâncime este definit într-un mod uşor diferit faţă de acela al unei căutări în lăţime: vom nota $G_\pi = (V, E_\pi)$, unde

$$E_\pi = \{(\pi[v], v) : v \in V \text{ şi } \pi[v] \neq \text{NIL}\}.$$

Subgraful predecesor al unei căutări în adâncime formează o ***pădure de adâncime*** compusă din mai mulţi ***arbori de adâncime***. Muchiile din E_π se numesc ***muchii de arbore***.

Ca în cazul căutării în lăţime, vârfurile sunt colorate în timpul căutării pentru a indica starea lor. Iniţial fiecare vârf este alb, pentru ca el să fie făcut, apoi, gri când este ***descoperit*** în timpul căutării, şi negru când este ***terminat***, adică atunci când lista lui de adiacenţă a fost examinată în întregime. Această tehnică garantează faptul că fiecare vârf ajunge exact într-un arbore de adâncime, astfel încât aceşti arbori sunt disjuncţi.

Pe lângă crearea unei păduri de adâncime, căutarea în adâncime creează pentru fiecare vârf şi ***marcaje de timp***. Fiecare vârf v are două astfel de marcaje: primul marcaj de timp $d[v]$ memorează momentul când v este descoperit pentru prima oară (şi colorat gri), iar al doilea marcaj $f[v]$ memorează momentul când căutarea termină de examinat lista de adiacenţă a lui v (şi îl colorează pe v în negru). Aceste marcaje de timp sunt folosite în mulţi algoritmi de grafuri şi sunt, în general, folositoare în argumentarea comportării căutării în adâncime.

Procedura CA de mai jos memorează în $d[u]$ momentul când u este descoperit şi în $f[u]$ momentul când u este terminat. Aceste marcaje de timp sunt valori întregi între 1 şi $2|V|$, deoarece există un eveniment de descoperire şi unul de terminare pentru fiecare din cele $|V|$ vârfuri. Pentru fiecare vârf u,

$$d[u] < f[u]. \tag{23.1}$$

Vârful u este ALB înainte de momentul $d[u]$, GRI între momentul $d[u]$ şi momentul $f[u]$ şi NEGRU după aceea.

Următorul pseudocod este algoritmul de bază al căutării în adâncime. Graful de intrare G poate fi orientat sau neorientat. Variabila *timp* este o variabilă globală pe care o folosim pentru aplicarea marcajelor de timp.

CA(G)
1: **pentru** fiecare vârf $u \in V[G]$ **execută**
2: $culoare[u] \leftarrow$ ALB
3: $\pi[u] \leftarrow$ NIL
4: $timp \leftarrow 0$
5: **pentru** fiecare vârf $u \in V[G]$ **execută**
6: **dacă** $culoare[u] =$ ALB **atunci**
7: CA-VIZITĂ(u)

CA-VIZITĂ(u)
1: $culoare[u] \leftarrow$ GRI ▷ Vârful alb u tocmai a fost descoperit.
2: $d[u] \leftarrow timp \leftarrow timp + 1$
3: **pentru** fiecare $v \in Adj[u]$ **execută** ▷ Explorează muchia (u, v).
4: **dacă** $culoare[v] =$ALB **atunci**
5: $\pi[v] \leftarrow u$
6: CA-VIZITĂ(v)
7: $culoare[u] \leftarrow$ NEGRU ▷ Vârful u este colorat în negru. El este terminat.
8: $f[u] \leftarrow timp \leftarrow timp + 1$

Figura 23.4 ilustrează execuţia lui CA pe graful prezentat în figura 23.2.

Procedura CA funcţionează după cum urmează. În liniile 1–3 se colorează toate vârfurile în alb şi se iniţializează câmpurile π ale lor cu NIL. În linia 4 contorul de timp global este resetat. În liniile 5–7 se verifică fiecare vârf din V căruia îi vine rândul şi, când este găsit un vârf alb, el este vizitat folosind CA-VIZITĂ. De fiecare dată când procedura CA-VIZITĂ(u) este apelată în linia 7, vârful u devine rădăcina unui nou arbore din pădurea de adâncime. Când CA-VIZITĂ se termină, fiecărui vârf u i-a fost atribuit un timp de descoperire $d[u]$ şi un timp de terminare $f[u]$.

În fiecare apel al procedurii CA-VIZITĂ(u), vârful u este iniţial alb. În linia 1 vârful u este colorat în gri, iar în linia 2 se memorează timpul de descoperire $d[u]$ incrementând şi salvând variabila globală *timp*. În liniile 3–6 este examinat fiecare vârf v, adiacent lui u, care este vizitat recursiv dacă acesta este alb. Pe măsură ce fiecare vârf $v \in Adj[u]$ este examinat în linia 3, spunem că muchia (u, v) este ***explorată*** de către căutarea în adâncime. În final, după ce fiecare muchie ce pleacă din u a fost explorată, în liniile 7–8 u este colorat în negru şi timpul de terminare este memorat în $f[u]$.

Care este timpul de execuţie al procedurii CA? Buclele din liniile 1–3 şi din liniile 5–7 ale procedurii CA necesită un timp $\Theta(V)$, pe lângă cel necesar executării apelurilor către CA-VIZITĂ. Procedura CA-VIZITĂ este apelată exact o dată pentru fiecare vârf $v \in V$, deoarece CA-VIZITĂ este invocată numai pentru vârfurile albe şi primul lucru pe care îl face este să coloreze fiecare vârf în gri. În timpul unei execuţii a CA-VIZITĂ(v), bucla din liniile 3–6 este executată de $|Adj[v]|$ ori. Deoarece

$$\sum_{v \in V} |Adj[v]| = \Theta(E),$$

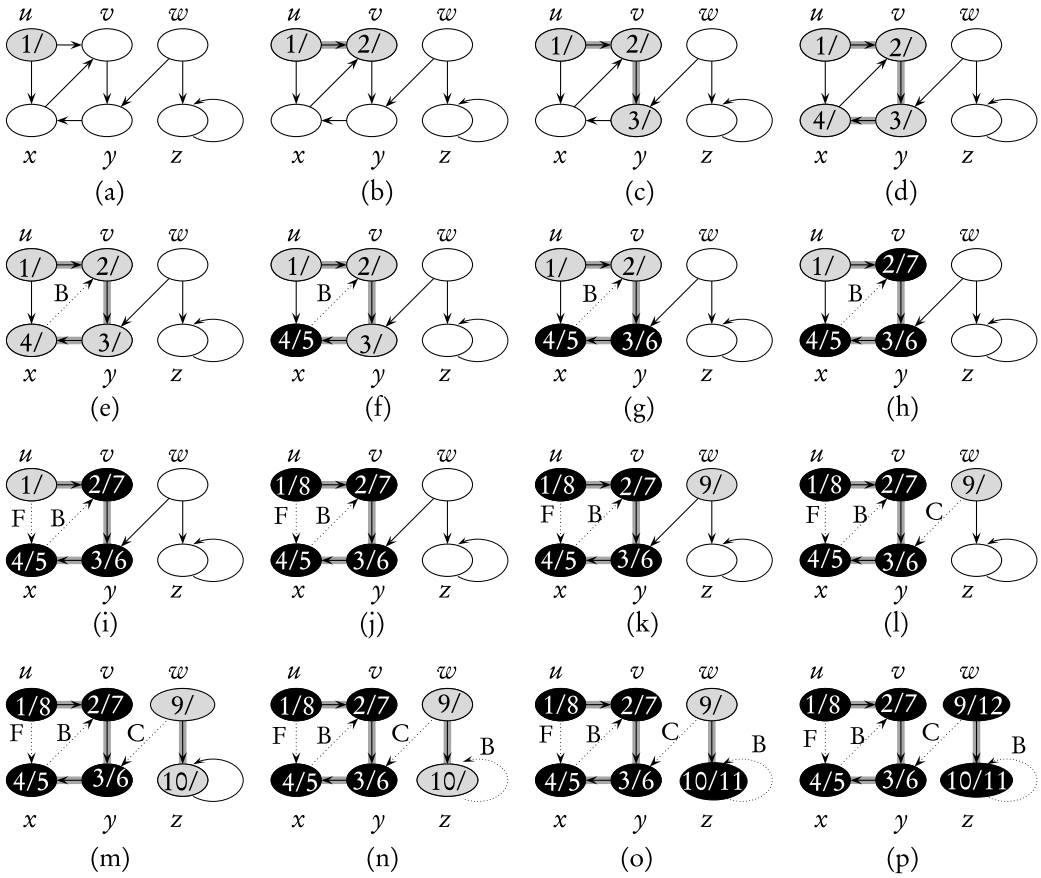


Figura 23.4 Progresul algoritmului de căutare în adâncime CA pe un graf orientat. Pe măsură ce muchiile sunt explorate de algoritm, ele sunt prezentate fie cu culoarea gri (dacă sunt muchii de arbori), fie cu linie punctată (altfel). Muchiile care nu aparţin arborilor sunt etichetate cu B, C, F, în funcţie de tipul lor (ele pot fi muchii înapoi, muchii transversale sau muchii înainte). Vârfurilor le sunt aplicate marcaje de timp, corespunzătoare timpului de descoperire, respectiv terminare.

costul total al execuţiei liniilor 3–6 ale CA-VIZITĂ este $\Theta(E)$. De aceea, timpul de execuţie al CA-VIZITĂ este $\Theta(V + E)$.

Proprietăţile căutării în adâncime

Căutarea în adâncime oferă multe informaţii despre structura unui graf. Proprietatea fundamentală a căutării în adâncime este că subgraful său predecesor G_π formează într-adevăr o pădure de arbori, deoarece structura arborilor de adâncime oglindeşte exact structura apelurilor recursive ale CA-VIZITĂ. Cu alte cuvinte, $u = \pi[v]$ dacă şi numai dacă CA-VIZITĂ(v) a fost apelată în timpul unei căutări în lista de adiacenţă a lui u.

O altă proprietate importantă a căutării în adâncime este că timpii de descoperire şi terminare au o ***structură de paranteză***. Dacă reprezentăm descoperirea unui vârf u printr-o paranteză

deschisă "(u" şi terminarea sa printr-o paranteză închisă "u)", atunci istoria descoperirilor şi a terminărilor formează o expresie corectă, în sensul că parantezele sunt corect împerecheate. De exemplu, căutarea în adâncime din figura 23.5(a) corespunde parantezării din figura 23.5(b). O altă cale de a enunţa condiţia structurii parantezelor este dată în următoarea teoremă:

Teorema 23.6 (Teorema parantezelor) În orice căutare în adâncime a unui graf (orientat sau neorientat) $G = (V, E)$, pentru orice două vârfuri u şi v, exact una din următoarele trei condiţii este adevărată:

- intervalele $[d[u], f[u]]$ şi $[d[v], f[v]]$ sunt total disjuncte,
- intervalul $[d[u], f[u]]$ este conţinut, în întregime, în intervalul $[d[v], f[v]]$, iar u este un descendent al lui v în arborele de adâncime, sau
- intervalul $[d[v], f[v]]$ este conţinut, în întregime, în intervalul $[d[u], f[u]]$, iar v este un descendent al lui u în arborele de adâncime.

Demonstraţie. Începem cu cazul în care $d[u] < d[v]$. În funcţie de valoarea de adevăr a inegalităţii $d[v] < f[u]$, există două subcazuri care trebuie considerate. În primul subcaz $d[v] < f[u]$, deci v a fost descoperit în timp ce u era încă gri. Aceasta implică faptul că v este un descendent al lui u. Mai mult, deoarece v a fost descoperit înaintea lui u, toate muchiile care pleacă din el sunt explorate, iar v este terminat, înainte ca algoritmul să revină pentru a-l termina pe u. De aceea, în acest caz, intervalul $[d[v], f[v]]$ este conţinut în întregime în intervalul $[d[u], f[u]]$. În celălalt subcaz $f[u] < d[v]$ şi inegalitatea (23.1) implică faptul că intervalele $[d[u], f[u]]$ şi $[d[v], f[v]]$ sunt disjuncte.

Cazul în care $d[v] < d[u]$ este similar, inversând rolurile lui u şi v în argumentaţia de mai sus. ∎

Corolarul 23.7 (Interclasarea intervalelor descendenţilor) Vârful v este un descendent normal al lui u în pădurea de adâncime pentru un graf G orientat sau neorientat dacă şi numai dacă $d[u] < d[v] < f[v] < f[u]$.

Demonstraţie. Este imediată din teorema 23.6. ∎

Următoarea teoremă oferă o altă caracterizare importantă a situaţiei în care un vârf este descendent al unui alt vârf în cadrul pădurii de adâncime.

Teorema 23.8 (Teorema drumului alb) Într-o pădure de adâncime a unui graf $G = (V, E)$ orientat sau neorientat, vârful v este un descendent al vârfului u dacă şi numai dacă la momentul $d[u]$, când căutarea îl descoperă pe u, vârful v este accesibil din u printr-un drum format în întregime din vârfuri albe.

Demonstraţie. $\Rightarrow$: Să presupunem că v este un descendent al lui u. Fie w un vârf oarecare pe drumul dintre u şi v în arborele de adâncime, astfel încât w este un descendent al lui u. Din corolarul 23.7, $d[u] < d[w]$, şi astfel w este alb la momentul $d[u]$.

$\Leftarrow$: Să presupunem că vârful v este accesibil din u printr-un drum alcătuit din vârfuri albe la momentul $d[u]$, dar v nu devine un descendent al lui u în arborele de adâncime. Fără a pierde din generalitate, vom presupune că fiecare din celelalte vârfuri din drumul respectiv devine un descendent al lui u. (Altfel, fie v cel mai apropiat vârf al lui u de pe drumul respectiv care nu devine un descendent al lui u.) Fie w predecesorul lui v de pe drum, astfel încât w

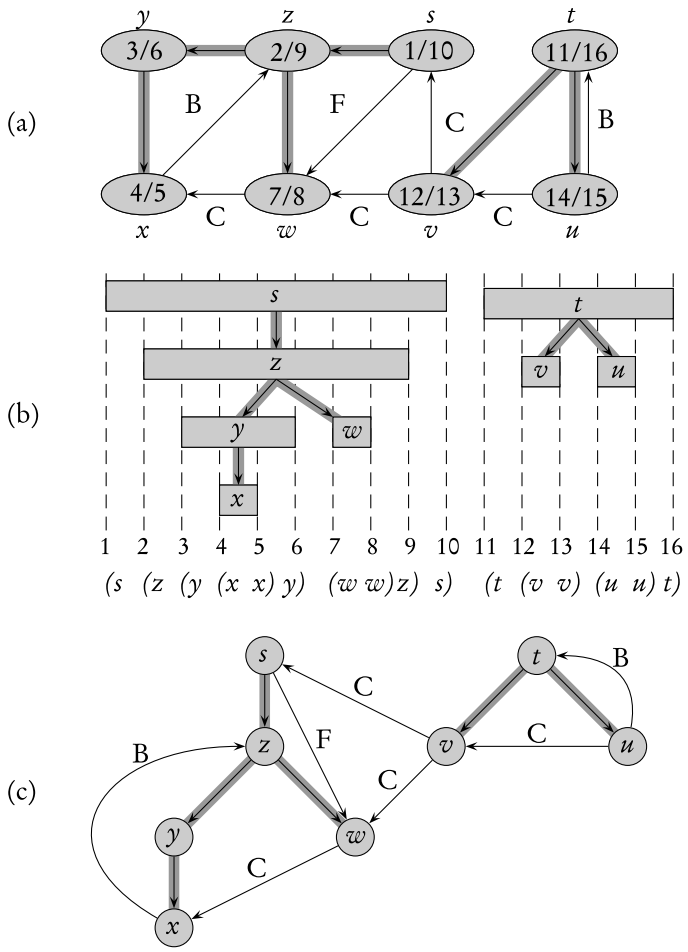


Figura 23.5 Proprietăţile căutării în adâncime. **(a)** Rezultatul căutării în adâncime pe un graf orientat. Vârfurilor li se ataşează marcaje de timp şi tipul muchiilor este indicat ca în figura 23.4. **(b)** Intervalele pentru timpii de descoperire şi de terminare ai fiecărui vârf corespund parantezării prezentate. Fiecare dreptunghi acoperă intervalul dat de timpii de descoperire şi terminare ai vârfului corespunzător. Muchiile de arbori sunt afişate. Dacă două intervale se suprapun, atunci unul este încadrat în celalalt, iar vârful corespunzător intervalului mai mic este un descendent al vârfului corespunzător celui mai mare. **(c)** Graful din partea (a) redesenat cu muchiile de arbori şi muchiile înainte mergând în jos în cadrul unei păduri de adâncime şi toate muchiile înapoi mergând în sus de la un descendent la un strămoş.

este un descendent al lui u (w şi u pot fi de fapt unul şi acelaşi vârf) şi, din corolarul 23.7, $f[w] \leq f[u]$. Să observăm că v trebuie descoperit după u, dar înainte ca w să fie terminat. De aceea, $d[u] < d[v] < f[w] \leq f[u]$. Atunci teorema 23.6 implică faptul că intervalul $[d[v], f[v]]$ este conţinut în întregime în intervalul $[d[u], f[u]]$. Din corolarul 23.7, v trebuie să fie un descendent al lui u. ■

Clasificarea muchiilor

O altă proprietate interesantă a căutării în adâncime este aceea că algoritmul de căutare poate fi folosit pentru clasificarea muchiilor grafului de intrare $G = (V, E)$. Această clasificare a muchiilor poate fi folosită pentru a afla informaţii importante despre un graf. De exemplu, în secţiunea următoare, vom vedea că un graf orientat este aciclic dacă şi numai dacă o căutare în adâncime nu produce nici o muchie "înapoi" (lema 23.10).

Putem defini patru tipuri de muchii în funcţie de pădurea de adâncime G_π produsă de o căutare în adâncime pe G.

1. ***Muchiile de arbore*** sunt muchii din pădurea de adâncime G_π. Muchia (u, v) este o muchie de arbore dacă v a fost descoperit explorând muchia (u, v).

2. ***Muchiile înapoi*** sunt acele muchii (u, v) care unesc un vârf u cu un strămoş v într-un arbore de adâncime. Buclele (muchii de la un vârf la el însuşi) care pot apărea într-un graf orientat sunt considerate muchii înapoi.

3. ***Muchiile înainte*** sunt acele muchii (u, v) ce nu sunt muchii de arbore şi conectează un vârf u cu un descendent v într-un arbore de adâncime.

4. ***Muchiile transversale*** sunt toate celelalte muchii. Ele pot uni vârfuri din acelaşi arbore de adâncime, cu condiţia ca unul să nu fie strămoşul celuilalt, sau pot uni vârfuri din arbori, de adâncime, diferiţi.

În figurile 23.4 şi 23.5, muchiile sunt etichetate pentru a indica tipul lor. Figura 23.5(c) arată, de asemenea, cum graful din figura 23.5(a) poate fi redesenat astfel încât toate muchiile de arbori şi cele înainte se îndreaptă în jos într-un arbore de adâncime şi toate muchiile înapoi merg în sus. Orice graf poate fi redesenat în această manieră.

Algoritmul CA poate fi modificat pentru a clasifica muchiile pe măsură ce le întâlneşte. Ideea principală este aceea că fiecare muchie (u, v) poate fi clasificată după culoarea vârfului v care este atins când muchia (u, v) este explorată pentru prima dată (cu excepţia faptului că nu se poate face distincţia între muchiile înainte şi cele transversale):

1. ALB indică o muchie de arbore,

2. GRI indică o muchie înapoi, şi

3. NEGRU indică o muchie înainte sau una transversală.

Primul caz este imediat din specificaţiile algoritmului. Pentru cel de-al doilea caz, se observă că vârfurile gri formează, întotdeauna, un lanţ liniar de descendenţi corespunzând stivei de apeluri active ale procedurii CA-VIZITĂ; numărul de vârfuri gri este cu unu mai mare decât adâncimea din pădurea de adâncime a vârfului celui mai recent descoperit. Explorarea începe întotdeauna din cel mai adânc vârf gri, deci o muchie care ajunge într-un alt vârf gri ajunge la un strămoş. Al treilea caz tratează posibilitatea rămasă. Se poate demonstra că o astfel de muchie (u, v) este o muchie înainte dacă $d[u] < d[v]$ şi o muchie transversală dacă $d[u] > d[v]$. (Vezi exerciţiul 23.3-4.)

Într-un graf neorientat poate exista o oarecare ambiguitate în clasificarea muchiilor, deoarece (u, v) şi (v, u) reprezintă, de fapt, aceeaşi muchie. În acest caz, muchia este clasificată ca fiind de *primul* tip din clasificare ale cărui condiţii sunt îndeplinite. În mod echivalent (vezi exerciţiul

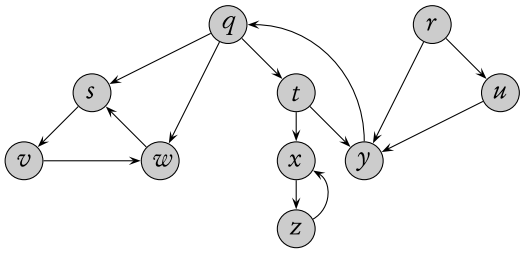


Figura 23.6 Un graf orientat pentru folosirea în exerciţiile 23.3-2 şi 23.5-2.

23.3-5), muchia este clasificată în funcţie de care din perechile (u, v) şi (v, u) apare prima în cursul execuţiei algoritmului.

Vom arăta acum că muchiile înainte şi cele transversale nu apar niciodată într-o căutare în adâncime într-un graf neorientat.

Teorema 23.9 Într-o căutare în adâncime într-un graf neorientat G, fiecare muchie a lui G este fie o muchie de arbore fie o muchie înapoi.

Demonstraţie. Fie (u, v) o muchie arbitrară a lui G şi să presupunem, fără a pierde din generalitate, că $d[u] < d[v]$. Atunci v trebuie descoperit şi terminat înainte ca u să fie terminat, deoarece v este pe lista de adiacenţă a lui u. Dacă muchia (u, v) este explorată prima dată în direcţia de la u la v, atunci (u, v) devine o muchie de arbore. Dacă (u, v) este explorată prima dată în direcţia de la v la u, atunci (u, v) este o muchie înapoi, deoarece u este încă gri în momentul când muchia este explorată pentru prima oară. ■

În următoarele secţiuni vom vedea multe aplicaţii ale acestor teoreme.

Exerciţii

23.3-1 Desenaţi un tabel de dimensiuni 3×3 şi etichetaţi liniile şi coloanele cu ALB, GRI şi NEGRU. În fiecare poziţie (i, j) indicaţi dacă, la vreun moment dat în timpul unei căutări în adâncime a unui graf orientat, poate exista o muchie de la un vârf de culoare i la un vârf de culoare j. Pentru fiecare muchie posibilă, indicaţi de ce tip poate fi ea. Faceţi un al doilea astfel de tabel pentru căutarea în adâncime într-un graf orientat.

23.3-2 Arătaţi cum funcţionează căutarea în adâncime pe graful din figura 23.6. Presupuneţi că bucla **pentru** din liniile 5–7 din procedura CA consideră vârfurile în ordine alfabetică şi că fiecare listă de adiacenţă este ordonată alfabetic. Determinaţi timpii de descoperire şi terminare pentru fiecare vârf şi prezentaţi clasificarea fiecărei muchii.

23.3-3 Specificaţi structura parantezelor determinată de căutarea în adâncime prezentată în figura 23.4.

23.3-4 Arătaţi că muchia (u, v) este

a. o muchie de arbore dacă şi numai dacă $d[u] < d[v] < f[v] < f[u]$,

b. o muchie înapoi dacă şi numai dacă $d[v] < d[u] < f[u] < f[v]$ şi

c. o muchie transversală dacă şi numai dacă $d[v] < f[v] < d[u] < f[u]$.

23.3-5 Arătaţi că, într-un graf neorientat, clasificarea unei muchii (u, v) ca fiind o muchie de arbore sau o muchie înapoi, pe baza faptului că muchia (u, v) este întâlnită înaintea muchiei (v, u) în timpul căutării în adâncime, este echivalentă cu clasificarea ei în funcţie de priorităţile tipurilor în schema de clasificare.

23.3-6 Daţi un contraexemplu pentru afirmaţia conform căreia, dacă într-un graf orientat G există un drum de la u la v şi într-o căutare în adâncime în G, $d[u] < d[v]$, atunci v este un descendent al lui u în pădurea de adâncime produsă.

23.3-7 Modificaţi pseudocodul pentru căutarea în adâncime, astfel încât să tipărească fiecare muchie într-un graf orientat G, împreună cu tipul acesteia. Arătaţi ce modificări trebuie făcute (în cazul în care trebuie făcută vreo modificare) dacă G este neorientat.

23.3-8 Explicaţi cum poate ajunge un vârf u al unui graf orientat într-un arbore de adâncime ce îl conţine numai pe u, deşi există în G atât muchii care intră în u cât şi muchii care ies din u.

23.3-9 Arătaţi că o căutare în adâncime, într-un graf neorientat G, poate fi folosită pentru a determina componentele conexe ale lui G, şi că numărul de arbori din pădurea de adâncime este egal cu numărul componentelor conexe ale lui G. Mai exact, arătaţi cum trebuie modificat algoritmul de căutare în adâncime, astfel încât fiecare vârf v este etichetat cu o valoare întreagă $cc[v]$ între 1 şi k, unde k este numărul de componente conexe ale lui G, astfel încât $cc[u] = cc[v]$ dacă şi numai dacă u şi v sunt în aceeaşi componentă conexă.

23.3-10 $\star$ Un graf orientat $G = (V, E)$ este ***unic conex*** dacă $u \rightsquigarrow v$ implică faptul că există cel mult un drum simplu de la u la v pentru toate vârfurile $u, v \in V$. Elaboraţi un algoritm eficient pentru a determina dacă un graf orientat este sau nu unic conex.

23.4. Sortarea topologică

Această secţiune prezintă modul în care căutarea în adâncime poate fi folosită pentru a executa sortări topologice ale unor grafuri orientate aciclice. O ***sortare topologică*** a unui graf orientat aciclic $G = (V, E)$ este o ordonare liniară a tuturor vârfurilor sale astfel încât, dacă G conţine o muchie (u, v), atunci u apare înaintea lui v în ordonare. (Dacă un graf nu este aciclic, atunci nu este posibilă nici o ordonare liniară). O sortare topologică a unui graf poate fi văzută ca o ordonare a vârfurilor sale de-a lungul unei linii orizontale, astfel încât toate muchiile sale orientate merg de la stânga la dreapta. Sortarea topologică este deci diferită de tipul normal de "ordonare" studiată în partea a II-a.

Grafurile orientate aciclice sunt folosite în multe aplicaţii pentru a indica precedenţa între evenimente. Figura 23.7 prezintă un exemplu care apare când profesorul Bumstead se îmbracă dimineaţa. Profesorul trebuie să îmbrace anumite lucruri înaintea altora (de exemplu, ciorapii înaintea pantofilor). O muchie orientată (u, v) din graful orientat aciclic din figura 23.7(a) indică faptul că articolul de îmbrăcăminte u trebuie îmbrăcat înaintea articolului v. De aceea, o sortare topologică a acestui graf orientat aciclic ne dă o ordine pentru îmbrăcare. Figura 23.7(b) prezintă

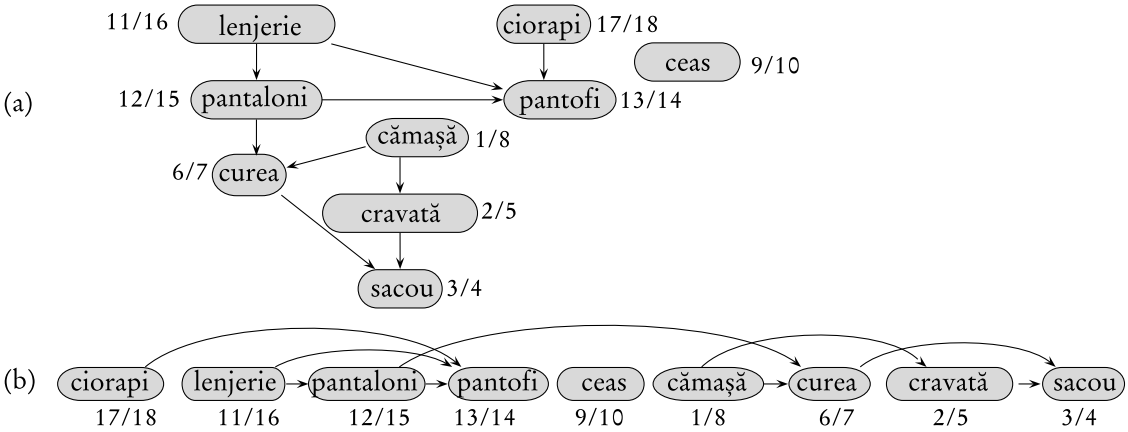


Figura 23.7 **(a)** Profesorul Bumstead îşi sortează topologic îmbrăcămintea când se îmbracă. Fiecare muchie (u, v) înseamnă că articolul u trebuie îmbrăcat înaintea articolului v. Timpii de descoperire şi de terminare dintr-o căutare în adâncime sunt prezentaţi alături de fiecare vârf. **(b)** Acelaşi graf sortat topologic. Vârfurile lui sunt aranjate de la stânga la dreapta în ordinea descrescătoare a timpului de terminare. Se observă că toate muchiile orientate merg de la stânga la dreapta.

graful orientat aciclic sortat topologic ca o ordonare a vârfurilor de-a lungul unei linii orizontale, astfel încât toate muchiile orientate merg de la stânga la dreapta.

Următorul algoritm simplu sortează topologic un graf orientat aciclic.

SORTARE-TOPOLOGICĂ(G)
1: apelează CA(G) pentru a calcula timpii de terminare $f[v]$ pentru fiecare vârf v
2: pe măsură ce fiecare vârf este terminat, inserează-l în capul unei liste înlănţuite
3: **returnează** lista înlănţuită de vârfuri

Figura 23.7(b) arată cum vârfurile sortate topologic apar în ordine inversă faţă de timpii lor de terminare.

Putem executa o sortare topologică într-un timp $\Theta(V + E)$, deoarece căutarea în adâncime necesită un timp $\Theta(V + E)$, iar inserarea fiecăruia din cele $|V|$ vârfuri în capul listei înlănţuite necesită un timp $O(1)$.

Vom demonstra corectitudinea acestui algoritm folosind următoarea lemă cheie, care caracterizează grafurile orientate aciclice.

Lema 23.10 Un graf orientat G este aciclic dacă şi numai dacă în urma unei căutări în adâncime în G, nu rezultă nici o muchie înapoi.

Demonstraţie. $\Rightarrow$: Să presupunem că există o muchie înapoi (u, v). Atunci vârful v este un strămoş al vârfului u în pădurea de adâncime. Există deci în G o cale de la v la u, iar muchia înapoi (u, v) completează un ciclu.

$\Leftarrow$: Să presupunem că G conţine un ciclu c. Vom arăta că, în urma unei căutări în adâncime, în G rezultă o muchie înapoi. Fie v primul vârf descoperit în c, şi fie (u, v) muchia precedentă în c. La momentul $d[v]$, există un drum de vârfuri albe de la v la u. Din teorema drumului alb

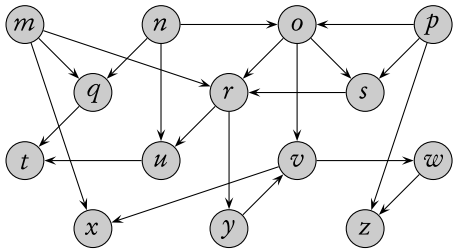


Figura 23.8 Un graf orientat aciclic pentru sortarea topologică.

rezultă că u devine un descendent al lui v în pădurea de adâncime. De aceea, (u, v) este o muchie înapoi. ■

Teorema 23.11 Procedura SORTARE-TOPOLOGICĂ(G) determină o sortare topologică a unui graf orientat aciclic G.

Demonstraţie. Să presupunem că algoritmul CA este rulat pe un graf orientat aciclic $G = (V, E)$ dat, pentru a determina timpul de terminare pentru vârfurile sale. Este suficient să arătăm că, pentru orice pereche de vârfuri distincte $u, v \in V$, dacă există o muchie în G de la u la v, atunci $f[v] < f[u]$. Să considerăm o muchie oarecare (u, v) explorată de procedura CA(G). Când această muchie este explorată, v nu poate fi gri deoarece v ar fi un strămoş al lui u şi (u, v) ar fi o muchie înapoi, contrazicând lema 23.10. De aceea, v trebuie să fie alb sau negru. Dacă v este alb, el devine un descendent al lui u şi, astfel, $f[v] < f[u]$. Dacă v este negru, atunci de asemenea $f[v] < f[u]$. Astfel, pentru orice muchie (u, v) din graful orientat aciclic, avem $f[v] < f[u]$, demonstrând teorema. ■

Exerciţii

23.4-1 Arătaţi ordinea vârfurilor produsă de procedura SORTARE-TOPOLOGICĂ atunci când aceasta este rulată pe graful orientat aciclic din figura 23.8, luând în considerare presupunerile de la exerciţiul 23.3-2.

23.4-2 Elaboraţi un algoritm care determină dacă un graf neorientat $G = (V, E)$ dat conţine sau nu un ciclu. Algoritmul propus trebuie să ruleze într-un timp $O(V)$, indiferent de $|E|$.

23.4-3 Demonstraţi sau contrademonstraţi că, dacă un graf orientat G conţine cicluri, atunci, procedura SORTARE-TOPOLOGICĂ(G) produce o ordine a vârfurilor care minimizează numărul de muchii "rele" care sunt în dezacord cu ordinea produsă.

23.4-4 O altă cale de a efectua o sortare topologică pe un graf orientat aciclic $G = (V, E)$ este de a găsi, în mod repetat, un vârf cu gradul interior egal cu 0 care este afişat, iar apoi eliminat din graf împreună cu toate muchiile care pleacă din el. Explicaţi cum se poate implementa această idee, astfel încât să ruleze într-un timp $O(V + E)$. Ce se întâmplă cu acest algoritm dacă G are cicluri?

23.5. Componente tare conexe

Vom considera acum o aplicaţie clasică a căutării în adâncime: descompunerea unui graf orientat în componentele sale tare conexe. Această secţiune prezintă un mod de a face această descompunere folosind două căutări în adâncime. Mulţi algoritmi care lucrează cu grafuri orientate încep cu o astfel de descompunere. Această abordare permite, de multe ori, divizarea problemei originale în mai multe subprobleme, una pentru fiecare componentă tare conexă. Combinarea soluţiilor subproblemelor urmează structura legăturilor dintre componentele tare conexe. Această structură poate fi reprezentată printr-un graf numit graful "componentelor", definit în exerciţiul 23.5-4.

Să ne reamintim, din capitolul 5, că o componentă tare conexă a unui graf orientat $G = (V, E)$ este o mulţime maximală de vârfuri $U \subseteq V$, astfel încât, pentru fiecare pereche de vârfuri u şi v din U, avem atât $u \rightsquigarrow v$ cât şi $v \rightsquigarrow u$. Cu alte cuvinte, vârfurile u şi v sunt accesibile unul din celălalt. În figura 23.9 este prezentat un exemplu.

Algoritmul nostru pentru găsirea componentelor tare conexe ale unui graf $G = (V, E)$ foloseşte transpusul lui G, care este definit în exerciţiul 23.1-3 ca fiind graful $G^T = (V, E^T)$, unde $E^T = \{(u, v) : (v, u) \in E\}$. Cu alte cuvinte, E^T este constituită din muchiile din G cu sensurile lor inversate. Dată fiind o reprezentare prin liste de adiacenţă a lui G, timpul necesar creării lui G^T este $O(V + E)$. Este interesant de observat că G şi G^T au exact aceleaşi componente tare conexe: u şi v sunt accesibile unul din celălalt în G dacă şi numai dacă sunt accesibile unul din celălalt în G^T. Figura 23.9(b) prezintă transpusul grafului din figura 23.9(a), cu componentele tare conexe colorate în gri.

Următorul algoritm de timp liniar (adică $\Theta(V + E)$) determină componentele tare conexe ale unui graf orientat $G = (V, E)$ folosind două căutări în adâncime, una în G şi una în G^T.

COMPONENTE-TARE-CONEXE(G)
1: apelează CA(G) pentru a calcula timpii de terminare $f[u]$ pentru fiecare vârf u
2: calculează G^T
3: apelează CA(G^T), dar în bucla principală a lui CA, consideră vârfurile în ordinea descrescătoare a timpilor $f[u]$ (calculaţi în linia 1)
4: afişează vârfurile fiecărui arbore în pădurea de adâncime din pasul 3 ca o componentă tare conexă separată

Acest algoritm, care arată foarte simplu, pare a nu avea nimic de-a face cu componentele tare conexe. În ceea ce urmează vom clarifica misterele alcătuirii sale şi îi vom demonstra corectitudinea. Începem cu două observaţii folositoare.

Lema 23.12 Dacă două vârfuri se află în aceeaşi componentă tare conexă, atunci nici un drum între ele nu părăseşte, vreodată, componenta tare conexă.

Demonstraţie. Fie u şi v două vârfuri din aceeaşi componentă tare conexă. Din definiţia componentei tare conexe, există drumuri de la u la v şi de la v la u. Fie vârful w de-a lungul unui drum $u \rightsquigarrow w \rightsquigarrow v$, astfel încât w este accesibil din u. Mai mult, deoarece există un drum $v \rightsquigarrow u$, ştim că u este accesibil din w prin drumul $w \rightsquigarrow v \rightsquigarrow u$. De aceea, u şi w fac parte din aceeaşi componentă conexă. Deoarece w a fost ales în mod arbitrar, teorema este demonstrată. ∎

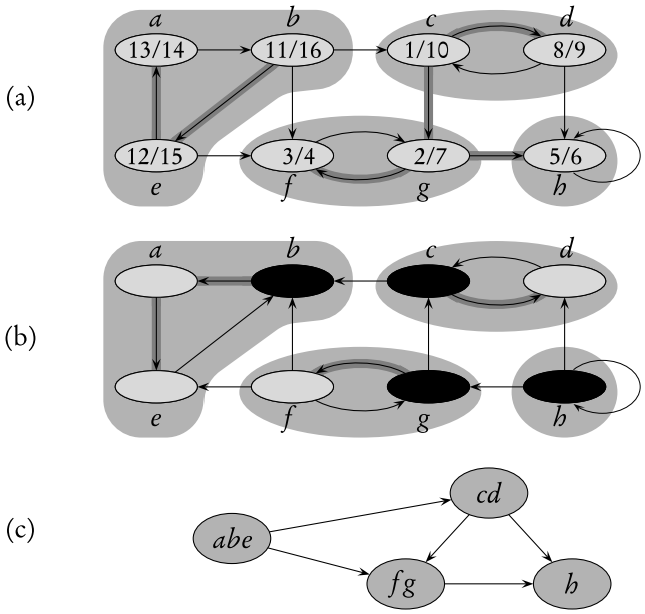


Figura 23.9 **(a)** Un graf orientat G. Componentele tare conexe ale lui G sunt prezentate în culoarea gri. Fiecare vârf este etichetat cu timpii săi de descoperire şi de terminare. Muchiile de arbore sunt prezentate în gri. **(b)**. Graful G^T, transpusul lui G. Este prezentat şi arborele de adâncime determinat în linia 3 a algoritmului COMPONENTE-TARE-CONEXE, cu muchiile de arbore în gri. Fiecare componentă tare conexă corespunde unui arbore de adâncime. Vârfurile *b, c, g* şi *h*, care sunt prezentate cu culoarea neagră, sunt strămoşii fiecărui vârf din componenta lor tare conexă; de asemenea, aceste vârfuri sunt rădăcinile arborilor de adâncime produşi de căutarea în adâncime în G^T. **(c)** Graful de componente aciclic G^{CTC} obţinut prin reducerea fiecărei componente tare conexe a lui G la un singur vârf.

Teorema 23.13 În orice căutare în adâncime, toate vârfurile din aceeaşi componentă tare conexă sunt situate în acelaşi arbore de adâncime.

Demonstraţie. Dintre vârfurile din componenta tare conexă, fie r primul descoperit. Deoarece r este primul, celelalte vârfuri din componenta tare conexă sunt albe în momentul când el este descoperit. Există drumuri de la r la toate celelalte vârfuri din componenta tare conexă; pentru că aceste drumuri nu părăsesc niciodată componenta tare conexă (din lema 23.12), toate vârfurile de pe ele sunt albe. Astfel, din teorema drumului alb, fiecare vârf din componenta tare conexă devine un descendent al lui r în componenta tare conexă. ∎

În restul acestei secţiuni, notaţiile $d[u]$ şi $f[u]$ se referă la timpii de descoperire şi terminare determinaţi de prima căutare în adâncime din linia 1 a algoritmului COMPONENTE-TARE-CONEXE. În mod similar, notaţia $u \rightsquigarrow v$ se referă la existenţa unui drum în G şi nu în G^T.

Pentru a demonstra că algoritmul COMPONENTE-TARE-CONEXE este corect, introducem noţiunea de ***predecesor*** $\phi(u)$ al unui vârf u, care este vârful w, accesibil din u, care e terminat ultimul în căutarea în adâncime din linia 1. Cu alte cuvinte,

$$\phi(u) = \text{acel vârf } w \text{ pentru care } u \rightsquigarrow w \text{ şi } f[w] \text{ este maximizat.}$$

Să observăm că relaţia $\phi(u) = u$ este posibilă, deoarece u este accesibil din el însuşi, şi de aici

$$f[u] \leq f[\phi(u)]. \tag{23.2}$$

De asemenea, putem demonstra că $\phi(\phi(u)) = \phi(u)$, deoarece, pentru oricare vârfuri $u, v \in V$,

$$u \rightsquigarrow v \text{ implică } f[\phi(v)] \leq f[\phi(u)], \tag{23.3}$$

deoarece $\{w : v \rightsquigarrow w\} \subseteq \{w : u \rightsquigarrow w\}$ şi predecesorul are timpul de terminare maxim dintre toate vârfurile accesibile. Deoarece $\phi(u)$ este accesibil din u, formula (23.3) implică faptul că $f[\phi(\phi(u))] \leq f[\phi(u)]$. Avem, de asemenea, $f[\phi(u)] \leq f[\phi(\phi(u))]$, din inegalitatea (23.2). Astfel, $f[\phi(\phi(u))] = f[\phi(u))]$, şi astfel obţinem $\phi(\phi(u)) = \phi(u)$, deoarece două vârfuri care sunt terminate în acelaşi moment reprezintă de fapt acelaşi vârf.

După cum vom vedea, fiecare componentă tare conexă are un vârf care este predecesorul fiecărui vârf din componenta tare conexă respectivă; acest predecesor este un "vârf reprezentativ" pentru acea componentă tare conexă. În căutarea în adâncime pe G, el este primul vârf al componentei tare conexe care este descoperit şi este ultimul vârf al acelei componente tare conexe care este terminat. În căutarea în adâncime a lui G^T, el este rădăcina unui arbore de adâncime. Vom demonstra acum aceste proprietăţi.

Prima teoremă justifică denumirea lui $\phi(u)$ de "predecesor" al lui u.

Teorema 23.14 Într-un graf orientat $G = (V, E)$, predecesorul $\phi(u)$ al oricărui vârf $u \in V$ din orice căutare în adâncime a lui G este un strămoş al lui u.

Demonstraţie. Dacă $\phi(u) = u$, teorema este evident adevărată. Dacă $\phi(u) \neq u$, vom considera culorile vârfurilor la momentul $d[u]$. Dacă $\phi(u)$ este negru, atunci $f[\phi(u)] < f[u]$, contrazicând inegalitatea (23.2). Dacă $\phi(u)$ este gri, atunci el este un strămoş al lui u, şi teorema este demonstrată.

Astfel ne rămâne de demonstrat că $\phi(u)$ nu este alb. Există două cazuri, în funcţie de culorile vârfurilor intermediare de pe drumul de la u la $\phi(u)$, dacă aceste vârfuri intermediare există.

1. Dacă fiecare vârf intermediar este alb, atunci $\phi(u)$ devine un descendent al lui u, din teorema drumului alb. Dar atunci $f[\phi(u)] < f[u]$, contrazicând inegalitatea (23.2).

2. Dacă vreun vârf intermediar nu este alb, fie t ultimul vârf care nu este alb de pe drumul de la u la $\phi(u)$. Atunci t trebuie să fie gri, deoarece nu există niciodată o muchie de la un vârf negru la unul alb, iar succesorul lui t este alb. Dar atunci există un drum de la t la $\phi(u)$ format din vârfuri albe, şi astfel $\phi(u)$ este un descendent al lui t, din teorema drumului alb. Aceasta implică faptul că $f[t] > f[\phi(u)]$, contrazicând alegerea lui $\phi(u)$, deoarece există un drum de la u la t. ∎

Corolarul 23.15 În orice căutare în adâncime într-un graf $G = (V, E)$, vârfurile u şi $\phi(u)$, pentru orice $u \in V$, fac parte din aceeaşi componentă tare conexă.

Demonstraţie. Avem $u \rightsquigarrow \phi(u)$, din definiţia predecesorului, şi $\phi(u) \rightsquigarrow u$, deoarece $\phi(u)$ este un strămoş al lui u. ∎

Următoarea teoremă oferă un rezultat mai puternic care face legătura între predecesori şi componentele tare conexe.

Teorema 23.16 Într-un graf orientat $G = (V, E)$, două vârfuri $u, v \in V$ fac parte din aceeaşi componentă tare conexă dacă şi numai dacă au acelaşi predecesor într-o căutare în adâncime în G.

Demonstraţie. $\Rightarrow$: Să presupunem că u şi v fac parte din aceeaşi componentă tare conexă. Fiecare vârf accesibil din u este accesibil din v şi invers, deoarece există drumuri în ambele direcţii între u şi v. Atunci, din definiţia predecesorului, tragem concluzia că $\phi(u) = \phi(v)$.

$\Leftarrow$: Să presupunem că $\phi(u) = \phi(v)$. Din corolarul 23.15, u face parte din aceeaşi componentă tare conexă ca $\phi(u)$, iar v face parte din aceeaşi componentă tare conexă ca $\phi(v)$. De aceea, u şi v fac parte din aceeaşi componentă conexă. ∎

Având teorema 23.16 la îndemână, structura algoritmului COMPONENTE-TARE-CONEXE poate fi înţeleasă mai uşor. Componentele tare conexe sunt mulţimi de vârfuri cu acelaşi predecesor. Mai mult, din teorema 23.14 şi din teorema parantezelor (teorema 23.6), în timpul căutării în adâncime în linia 1 a algoritmului COMPONENTE-TARE-CONEXE un predecesor este atât primul vârf descoperit cât şi ultimul vârf terminat din componenta lui tare conexă.

Pentru a înţelege de ce facem căutarea în adâncime în G^T din linia 3 a algoritmului COMPONENTE-TARE-CONEXE, vom considera vârful r cu cel mai mare timp de terminare calculat de căutarea în adâncime din linia 1. Din definiţia predecesorului, vârful r trebuie să fie un predecesor, deoarece este propriul său predecesor: este accesibil din el însuşi, şi nici un alt vârf din graf nu are un timp de terminare mai mare. Care sunt celelalte vârfuri din componenta tare conexă a lui r? Sunt acele vârfuri care îl au pe r drept predecesor – acelea din care r este accesibil, dar nici un vârf cu un timp de terminare mai mare decât $f[r]$ nu este accesibil din ele. Dar timpul de terminare al lui r este maxim dintre toate vârfurile din G. Din această cauză, componenta tare conexă a lui r este constituită din acele vârfuri din care r este accesibil. În mod echivalent, *componenta tare conexă a lui r constă din vârfurile accesibile din r în G^T.* Astfel, căutarea în adâncime din linia 3 identifică toate vârfurile din componenta tare conexă a lui r şi le face negre. (O căutare în lăţime, sau *oricare* căutare a vârfurilor accesibile, ar fi putut identifica această mulţime la fel de uşor.)

După ce căutarea în adâncime din linia 3 este efectuată, identificând componenta tare conexă a lui r, algoritmul începe cu vârful r' cu cel mai mare timp de terminare, care nu se află în componenta tare conexă a lui r. Vârful r' trebuie să fie propriul lui predecesor, deoarece nici un alt vârf cu un timp de terminare mai mare nu este accesibil din el (altfel, ar fi fost inclus în componenta tare conexă a lui r). Printr-un raţionament similar, orice vârf din care r' este accesibil şi care nu este deja negru, trebuie să fie în componenta tare conexă a lui r'. Astfel, pe măsură ce căutarea în adâncime din linia 3 continuă, algoritmul identifică şi înnegreşte fiecare vârf din componenta tare conexă a lui r' căutând din r' în G^T.

Astfel, căutarea în adâncime din linia 3 "decojeşte" componentele tare conexe una câte una. Fiecare componentă este identificată în linia 7 a CA printr-un apel către procedura CA-VIZITĂ cu predecesorul componentei ca argument. Apelurile recursive din cadrul CA-VIZITĂ înnegresc, în cele din urmă, fiecare vârf din componentă. Când CA-VIZITĂ revine în CA, întreaga componentă este înnegrită şi "decojită". Apoi CA găseşte vârful cu timpul de terminare maxim care n-a fost încă înnegrit; acest vârf este predecesorul unei alte componente, şi procedeul continuă.

Următoarea teoremă formalizează această argumentaţie.

Teorema 23.17 Algoritmul COMPONENTE-TARE-CONEXE(G) calculează corect componentele tare conexe ale unui graf G.

Demonstraţie. Vom argumenta prin inducţie, după numărul de arbori de adâncime găsiţi în timpul căutării în adâncime a lui G^T, că vârfurile din fiecare arbore formează o componentă tare conexă. Fiecare pas al demonstraţiei inductive demonstrează că un arbore format în timpul căutării în adâncime a lui G^T este o componentă tare conexă, presupunând că toţi arborii produşi înainte sunt componente tare conexe. Baza pentru această inducţie este trivială, din moment ce pentru primul arbore rezultat nu există arbori anteriori, şi de aici presupunerea este evident adevărată.

Să considerăm un arbore de adâncime T cu rădăcina r rezultat în timpul căutării în adâncime a lui G^T. Vom nota cu $C(r)$ mulţimea vârfurilor cu strămoşul r:

$$C(r) = \{v \in V : \phi(v) = r\}.$$

Vom demonstra acum că un vârf u este plasat în T dacă şi numai dacă $u \in C(r)$.

$\Leftarrow$: Teorema 23.13 implică faptul că fiecare vârf din $C(r)$ ajunge în acelaşi arbore de adâncime. Deoarece $r \in C(r)$ şi r este rădăcina lui T, fiecare element al lui $C(r)$ ajunge în T.

$\Rightarrow$: Vom arăta că orice vârf w, astfel încât $f[\phi(w)] > f[r]$ sau $f[\phi(w)] < f[r]$, nu este plasat în T, considerând aceste două cazuri separat. Prin inducţie după numărul de arbori găsiţi, orice vârf w, astfel încât $f[\phi(w)] > f[r]$, nu este plasat în arborele T, deoarece, la momentul când r este selectat, w va fi fost deja pus în arborele cu rădăcina $\phi(w)$. Orice vârf w, astfel încât $f[\phi(w)] < f[r]$, nu poate fi plasat în T, deoarece o astfel de plasare ar implica $w \rightsquigarrow r$. Astfel, din formula (23.3) şi din proprietatea că $r = \phi(r)$, obţinem că $f[\phi(w)] \geq f[\phi(r)] = f[r]$, ceea ce contrazice $f[\phi(w)] < f[r]$.

De aceea, T conţine doar acele vârfuri u pentru care $\phi(u) = r$. Cu alte cuvinte, T este identic cu componenta tare conexă $C(r)$, ceea ce completează demonstraţia inductivă. ∎

Exerciţii

23.5-1 Cum se schimbă numărul componentelor tare conexe ale unui graf dacă este adăugată o nouă muchie?

23.5-2 Arătaţi cum lucrează procedura COMPONENTE-TARE-CONEXE pe graful din figura 23.6. Mai exact, determinaţi timpii de terminare calculaţi în linia 1 şi pădurea produsă de linia 3. Presupuneţi că bucla din liniile 5–7 a CA consideră vârfurile în ordine alfabetică şi că listele de adiacenţă sunt în ordine alfabetică.

23.5-3 Profesorul Deaver pretinde că algoritmul pentru determinarea componentelor tare conexe poate fi simplificat, folosind graful original (în locul celui transpus) în cea de-a doua căutare în adâncime şi parcurgând vârfurile în ordinea *crescătoare* a timpilor de terminare. Are dreptate profesorul?

23.5-4 Notăm ***graful componentelor*** lui $G = (V, E)$ cu $G^{CTC} = (V^{CTC}, E^{CTC})$, unde V^{CTC} conţine un vârf pentru fiecare componentă tare conexă a lui G şi E^{CTC} conţine o muchie (u, v) dacă există o muchie orientată de la un vârf din componenta tare conexă a lui G corespunzătoare lui u la un vârf din componenta tare conexă a lui G corespunzătoare lui v. În figura 23.9(c) este prezentat un exemplu. Demonstraţi că G^{CTC} este un graf orientat aciclic.

23.5-5 Elaboraţi un algoritm care să ruleze într-un timp $O(V + E)$ şi care să calculeze graful componentelor unui graf orientat $G = (V, E)$. Asiguraţi-vă că există, cel mult, o muchie între două vârfuri în graful componentelor pe care îl determină algoritmul vostru.

23.5-6 Dat fiind un graf orientat $G = (V, E)$, explicaţi cum se poate crea un alt graf $G' = (V, E')$, astfel încât (a) G' are aceleaşi componente tare conexe ca G, (b) G' are acelaşi graf al componentelor ca G, şi (c) E' este cât mai mică posibil. Elaboraţi un algoritm rapid pentru determinarea lui G'.

23.5-7 Un graf orientat $G = (V, E)$ se numeşte ***semiconex*** dacă, pentru toate perechile de vârfuri $u, v \in V$, avem $u \rightsquigarrow v$ sau $v \rightsquigarrow u$. Elaboraţi un algoritm eficient pentru a determina dacă G este sau nu semiconex. Demonstraţi că algoritmul vostru este corect şi analizaţi timpul său de execuţie.

Probleme

23-1 Clasificarea muchiilor în funcţie de căutarea în lăţime
O pădure de adâncime clasifică muchiile unui graf în muchii de arbore, înapoi, înainte şi transversale. Un arbore de lăţime poate fi folosit, de asemenea, pentru a clasifica muchiile accesibile din sursa căutării în aceleaşi patru categorii.

a. Demonstraţi că într-o căutare în lăţime într-un graf neorientat, următoarele proprietăţi sunt adevărate:

1. Nu există muchii înapoi sau muchii înainte.
2. Pentru orice muchie (u, v), avem $d[v] = d[u] + 1$.
3. Pentru orice muchie transversală (u, v), avem $d[v] = d[u]$ sau $d[v] = d[u] + 1$.

b. Demonstraţi că într-o căutare în lăţime într-un graf orientat, următoarele proprietăţi sunt adevărate:

1. Nu există muchii înainte.
2. Pentru orice muchie de arbore (u, v), avem $d[v] = d[u] + 1$.
3. Pentru orice muchie transversală (u, v), avem $d[v] \leq d[u] + 1$.
4. Pentru orice muchie înapoi (u, v), avem $0 \leq d[v] < d[u]$.

23-2 Puncte de articulaţie, punţi şi componente biconexe
Fie $G = (V, E)$ un graf neorientat, conex. Un ***punct de articulaţie*** al lui G este un vârf a cărui eliminare face ca G să-şi piardă proprietatea de conexitate. O ***punte*** este o muchie a cărei eliminare face ca G să-şi piardă proprietatea de conexitate. O ***componentă biconexă*** a lui G este o mulţime maximală de muchii, astfel încât oricare două muchii fac parte dintr-un ciclu simplu comun. Figura 23.10 ilustrează aceste definiţii. Putem determina puncte de articulaţie, punţi şi componente biconexe folosind căutarea în adâncime. Fie $G_\pi = (V, E_\pi)$ un arbore de adâncime al lui G.

a. Demonstraţi că rădăcina lui G_π este un punct de articulaţie al lui G dacă şi numai dacă are cel puţin doi descendenţi în G_π.

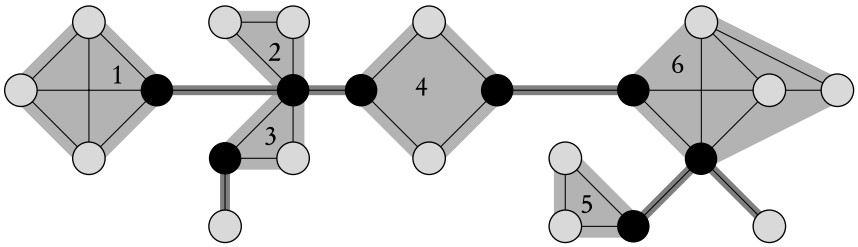


Figura 23.10 Punctele de articulaţie, punţile şi componentele biconexe ale unui graf neorientat, conex, pentru a fi folosite în problema 23-2. Punctele de articulaţie sunt vârfurile de culoare neagră, punţile sunt muchiile de culoare gri închis, iar componentele biconexe sunt muchiile din zonele de culoare gri deschis, împreună cu care este prezentată şi o numerotare a lor.

b. Fie v un vârf al lui G_π care nu este rădăcina acestuia. Demonstraţi că v este un punct de articulaţie al lui G dacă şi numai dacă v are un descendent s, astfel încât nu există nici o muchie înapoi de la s sau orice descendent al lui s la un strămoş nedegenerat al lui v.

c. Fie

$$jos[v] = min \left\{ \begin{array}{ll} d[v], & \\ \{d[w] : & (u,w) \text{ este o muchie înapoi pentru un descendent} \\ & \text{oarecare } u \text{ al lui } v\}. \end{array} \right.$$

Arătaţi cum se poate calcula $jos[v]$ pentru toate vârfurile $v \in V$ într-un timp $O(E)$.

d. Arătaţi cum se pot determina toate punctele de articulaţie într-un timp $O(E)$.

e. Demonstraţi că o muchie a lui G este o punte dacă şi numai dacă nu face parte dintr-un ciclu simplu al lui G.

f. Arătaţi cum se pot determina toate punţile lui G într-un timp $O(E)$.

g. Demonstraţi următoarea afirmaţie: Componentele biconexe ale lui G partiţionează muchiile lui G care nu sunt punţi.

h. Elaboraţi un algoritm care să ruleze într-un timp $O(E)$ care să eticheteze fiecare muchie e a lui G cu un număr pozitiv $cb[e]$ astfel încât $cb[e] = cb[e']$ dacă şi numai dacă e şi e' fac parte din aceeaşi componentă biconexă.

23-3 Tur Euler

Un ***tur Euler*** al unui graf orientat, conex $G = (V, E)$ este un ciclu care traversează fiecare muchie a lui G exact o dată, deşi s-ar putea ca un vârf să fie vizitat de mai multe ori.

a. Arătaţi că G are un tur Euler dacă şi numai dacă grad-interior(v)=grad-exterior(v) pentru orice vârf $v \in V$.

b. Elaboraţi un algoritm care să ruleze într-un timp $O(E)$ şi care să determine un tur Euler al lui G dacă el există. (*Indicaţie:* uniţi ciclurile cu muchii disjuncte.)

Note bibliografice

Even [65] şi Tarjan [188] constituie referinţe excelente pentru algoritmi de grafuri.

Căutarea în lăţime a fost descoperită de către Moore [150] în contextul căutării de drumuri în labirinturi. Lee [134] a descoperit, în mod independent, acelaşi algoritm în contextul stabilirii traseelor firelor de pe plăcile de circuite.

Hopcroft şi Tarjan [102] au argumentat folosirea reprezentării prin liste de adiacenţă în defavoarea reprezentării prin matrice de adiacenţă, pentru grafurile rare, şi au fost primii care au recunoscut importanţa algoritmică a căutării în adâncime. Căutarea în adâncime a fost folosită pe scară largă începând cu sfârşitul anilor '50, în special în programele din domeniul inteligenţei artificiale.

Tarjan [185] a elaborat un algoritm liniar pentru determinarea componentelor tare conexe. Algoritmul pentru componentele tare conexe din secţiunea 23.5 este adaptat din Aho, Hopcroft şi Ullman [5], care îl atribuie lui S. R. Kosaraju şi M. Sharir. Knuth [121] a fost primul care a dat un algoritm liniar pentru sortarea topologică.

24 Arbori de acoperire minimi

În proiectarea circuitelor electronice, este de multe ori necesar să facem pinii mai multor componente echivalenţi electronic, conectându-i împreună. Pentru interconectarea unei mulţimi de n pini, putem folosi un aranjament de $n-1$ fire, fiecare conectând doi pini. Dintre toate aceste aranjamente, cel care foloseşte cantitatea cea mai mică de fir este de obicei cel mai dezirabil.

Putem modela această problemă a conectării folosind un graf neorientat, conex $G = (V, E)$, unde V este mulţimea pinilor, iar E este mulţimea interconectărilor posibile între perechile de pini, şi, pentru fiecare pereche de pini $(u, v) \in E$, avem un cost $w(u, v)$ specificând cantitatea de fir necesară pentru conectarea lui u şi v. Dorim, apoi, să găsim o submulţime aciclică $T \subseteq E$ care conectează toate vârfurile şi al cărei cost total

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

este minimizat. Deoarece mulţimea T este aciclică şi conectează toate vârfurile, ea trebuie să formeze un arbore, pe care îl vom numi ***arbore de acoperire***[1] deoarece "acoperă" graful G. Problema determinării arborelui T o vom numi ***problema arborelui de acoperire minim.***[2] Figura 24.1 prezintă un exemplu de graf conex împreună cu arborele său minim de acoperire.

În acest capitol vom examina doi algoritmi pentru rezolvarea problemei arborelui parţial de acoperire minim: algoritmul lui Kruskal şi algoritmul lui Prim. Fiecare poate fi determinat uşor să ruleze într-un timp de ordinul $O(E \lg V)$ folosind heap-uri binare obişnuite. Folosind heap-uri Fibonacci, algoritmul lui Prim poate fi optimizat pentru a rula într-un timp de ordinul $O(E + V \lg V)$, care este o îmbunătăţire dacă $|V|$ este mult mai mic decât $|E|$.

Cei doi algoritmi ilustrează, de asemenea, o metodă euristică pentru optimizare, numită strategia "greedy". La fiecare pas al algoritmului, trebuie făcută una dintre alegerile posibile. Strategia greedy susţine faptul că trebuie făcută alegerea care este cea mai bună la momentul dat. În general, o astfel de strategie nu garantează obţinerea unei soluţii optime global pentru o anumită problemă. Totuşi, pentru problema arborelui de acoperire minim, putem demonstra că anumite strategii greedy produc un arbore de acoperire de cost minim. Strategiile greedy sunt discutate pe larg în capitolul 17. Deşi prezentul capitol poate fi citit independent de capitolul 17, metodele greedy prezentate în acest capitol sunt aplicaţii clasice ale noţiunilor teoretice introduse acolo.

Secţiunea 24.1 introduce un algoritm "generic" de arbore de acoperire minim, care dezvoltă arborele de acoperire adăugând o muchie o dată. Secţiunea 24.2 prezintă două moduri de implementare a algoritmului generic. Primul algoritm, datorat lui Kruskal, este similar cu algoritmul de componente conexe din secţiunea 22.1. Cel de-al doilea algoritm, datorat lui Prim, este similar cu algoritmul lui Dijkstra pentru drumuri de lungime minimă (secţiunea 25.2).

[1] În literatura de specialitate în limba română se foloseşte uneori şi termenul de "arbore parţial" – n. t.

[2] Expresia "arbore de acoperire minim" este o formă prescurtată pentru "arbore de acoperire de cost minim". De exemplu, nu minimizăm numărul de muchii din T, deoarece toţi arborii de acoperire au exact $|V| - 1$ muchii, conform teoremei 5.2.

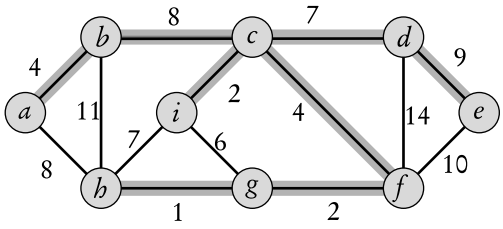


Figura 24.1 Un arbore de acoperire minim pentru un graf conex. Sunt prezentate costurile muchiilor, iar muchiile arborelui de acoperire minim sunt haşurate cu gri. Costul total al arborelui prezentat este 37. Arborele nu este unic: dacă eliminăm muchia (b, c) şi o înlocuim cu muchia (a, h), obţinem un alt arbore de acoperire de cost 37.

24.1. Dezvoltarea unui arbore de acoperire minim

Să presupunem că avem un graf neorientat, conex $G = (V, E)$ cu o funcţie de cost $w : E \to \mathbb{R}$ şi dorim să găsim un arbore de acoperire minim pentru G. Cei doi algoritmi, pe care îi prezentăm în acest capitol, folosesc o abordare greedy a problemei, deşi diferă prin modul în care implementează această abordare.

Această strategie greedy este implementată de următorul algoritm "generic", care dezvoltă arborele de acoperire minim adăugând o muchie o dată. Algoritmul foloseşte o mulţime A care este întotdeauna o submulţime a unui arbore de acoperire minim. La fiecare pas este determinată o muchie (u, v), care poate fi adăugată la A, respectând proprietatea de mai sus, în sensul că $A \cup \{(u, v)\}$ este, de asemenea, o submulţime a unui arbore de acoperire minim. Numim o astfel de muchie o ***muchie sigură*** pentru A, deoarece poate fi adăugată, în siguranţă, mulţimii A, respectând proprietatea de mai sus.

AAM-GENERIC(G, w)
1: $A \leftarrow \emptyset$
2: **cât timp** A nu formează un arbore de acoperire **execută**
3: găseşte o muchie (u, v) care este sigură pentru A
4: $A \leftarrow A \cup \{(u, v)\}$
5: **returnează** A

Observaţi că, după linia 1, mulţimea A satisface, într-un mod trivial, condiţia ca ea să fie o submulţime a arborelui de acoperire minim. Bucla din liniile 2–4 menţine condiţia. De aceea, când mulţimea A este returnată în linia 5, ea trebuie să fie un arbore minim de acoperire. Partea mai dificilă este, desigur, găsirea unei muchii sigure în linia 3. Una trebuie să existe, deoarece, atunci când linia 3 este executată, condiţia dictează faptul că există un arbore de acoperire minim T, astfel încât $A \subseteq T$, şi, dacă există o muchie $(u, v) \in T$, astfel încât $(u, v) \notin A$, atunci (u, v) este sigură pentru A.

În ceea ce urmează, vom da o regulă (teorema 24.1) pentru a recunoaşte muchiile sigure. Următoarea secţiune descrie doi algoritmi care folosesc această regulă pentru a găsi eficient muchiile sigure.

În primul rând, avem nevoie de câteva definiţii. O ***tăietură*** $(S, V - S)$ a unui graf neorientat

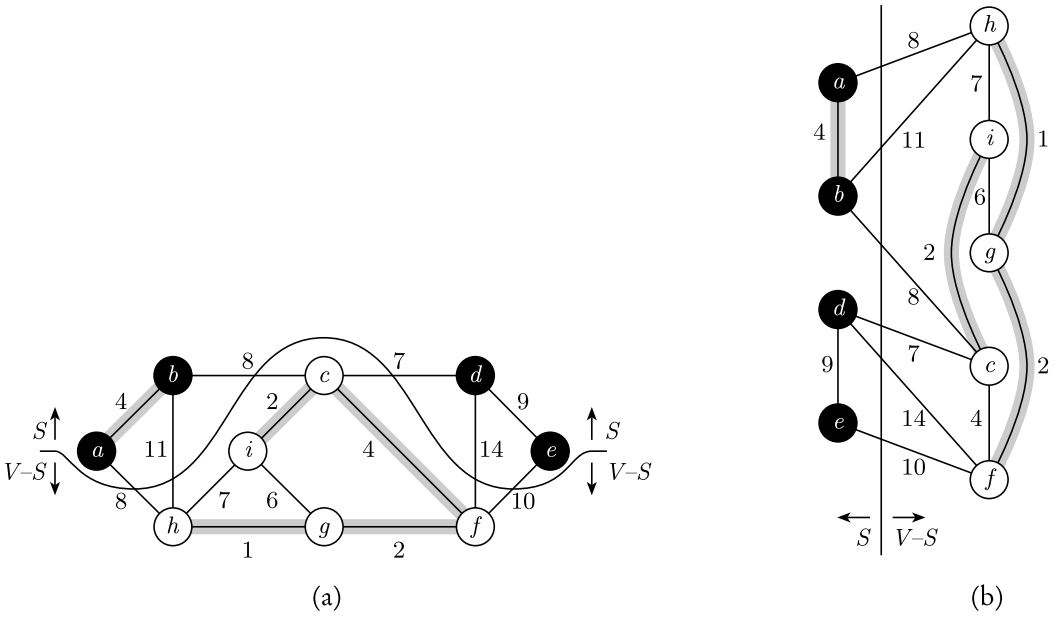


Figura 24.2 Două moduri de a privi o tăietură $(S, V - S)$ a grafului din figura 24.1. **(a)** Vârfurile din mulţimea S sunt colorate cu negru, iar cele din $V - S$ cu alb. Muchiile care traversează tăietura sunt cele care unesc vârfurile albe cu cele negre. Muchia (d, c) este unica muchie uşoară care traversează tăietura. O submulţime de muchii A este haşurată cu gri. Observaţi că tăietura $(S, V - S)$ respectă mulţimea A, deoarece nici o muchie din A nu traversează tăietura. **(b)** Acelaşi graf cu vârfurile din mulţimea S dispuse pe partea stângă şi cele din mulţimea $V - S$ în dreapta. O muchie traversează tăietura dacă uneşte un vârf din stânga cu un vârf din dreapta.

$G = (V, E)$ este o partiţie a lui V. Figura 24.2 ilustrează această noţiune. Spunem că o muchie $(u, v) \in E$ ***traversează*** tăietura $(S, V - S)$ dacă unul din punctele sale terminale este în S, şi celălalt în $V - S$. Spunem că o tăietură ***respectă*** mulţimea de muchii A dacă nici o muchie din A nu traversează tăietura. O muchie este o ***muchie uşoară*** care traversează o tăietură dacă are costul minim dintre toate muchiile care traversează tăietura. Observaţi că pot exista mai multe muchii uşoare care traversează o tăietură în cazul în care ele au costuri egale. Generalizând, spunem că o muchie este o ***muchie uşoară*** care satisface o proprietate dată dacă are costul minim dintre toate muchiile care satisfac acea proprietate.

Regula noastră pentru recunoaşterea muchiilor sigure este dată de următoarea teoremă.

Teorema 24.1 Fie $G = (V, E)$ un graf neorientat, conex cu o funcţie de cost w cu valori reale, definită pe E. Fie A o submulţime a lui E care este inclusă într-un arbore de acoperire minim al lui G, fie $(S, V - S)$ orice tăietură a lui G care respectă mulţimea A şi fie (u, v) o muchie uşoară care travesează tăietura $(S, V - S)$. Atunci muchia (u, v) este sigură pentru A.

Demonstraţie. Fie T un arbore minim de acoperire care include mulţimea A. Să presupunem că T nu conţine muchia uşoară (u, v), deoarece, dacă o conţine, demonstraţia este gata. Vom construi un alt arbore minim de acoperire T' care include mulţimea $A \cup \{(u, v)\}$ folosind o tehnică de tăiere şi lipire, arătând astfel că (u, v) este o muchie sigură pentru A.

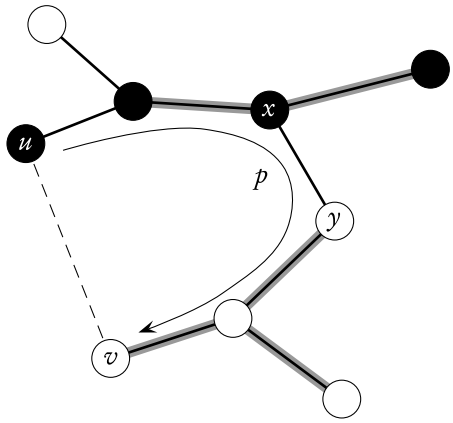


Figura 24.3 Demonstraţia teoremei 24.1. Vârfurile din S sunt negre, iar cele din $V - S$ sunt albe. Muchiile din arborele de acoperire minim T sunt prezentate, însă muchiile din graful G nu. Muchiile din A sunt haşurate cu gri, iar (u, v) este o muchie uşoară care traversează tăietura $(S, V - S)$. Muchia (x, y) este o muchie pe drumul unic p, de la u la v, din arborele T. Un arbore minim de acoperire T' care conţine (u, v) este format eliminând muchia (x, y) din T şi adăugând muchia (u, v).

Muchia (u, v) formează un ciclu cu muchiile de pe drumul p, de la u la v, din arborele T, după cum se vede în figura 24.3. Deoarece u şi v se află în părţi opuse ale tăieturii $(S, V - S)$, există, cel puţin, o muchie din T pe drumul p, care traversează, de asemenea, tăietura. Fie (x, y) orice astfel de muchie. Muchia (x, y) nu se află în A pentru că tăietura respectă mulţimea A. Deoarece (x, y) se află pe unicul drum de la u la v din arborele T, eliminarea lui (x, y) îl rupe pe T în două componente. Adăugarea muchiei (u, v) le reconectează pentru a forma un nou arbore de acoperire $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

Vom arăta, acum, că T' este un arbore de acoperire minim. Deoarece (u, v) este o muchie uşoară care traversează $(S, V - S)$ şi (x, y) traversează, de asemenea, această tăietură, rezultă că $w(u, v) \leq w(x, y)$. De aceea,

$$w(T') \quad = \quad w(T) - w(x, y) + w(u, v) \quad \leq \quad w(T).$$

Dar T este un arbore de acoperire minim, astfel încât $w(T) \leq w(T')$; astfel, T' este, de asemenea, un arbore de acoperire minim.

Mai rămâne de arătat că (u, v) este, într-adevăr, o muchie sigură pentru A. Avem $A \subseteq T'$, deoarece $A \subseteq T$ şi $(x, y) \notin A$. Astfel, $A \cup \{(u, v)\} \subseteq T'$. În concluzie, deoarece T' este un arbore minim de acoperire, (u, v) este o muchie sigură pentru A. ∎

Teorema 24.1 ne ajută să înţelegem, mai bine, modul de lucru al algoritmului AAM-Generic pe un graf conex $G = (V, E)$. Când algoritmul începe, mulţimea A este întotdeauna aciclică. Altfel, un arbore de acoperire minim care îl include pe A ar conţine un ciclu, ceea ce este o contradicţie. În orice moment al execuţiei algoritmului, graful $G_A = (V, A)$ este o pădure şi fiecare din componentele conexe ale lui G_A este un arbore. (Unii arbori pot conţine doar un singur vârf, cum se întâmplă în momentul în care algoritmul începe: A este vidă şi pădurea conţine $|V|$ arbori, câte unul pentru fiecare vârf.) Mai mult, orice muchie (u, v) sigură pentru A uneşte componentele distincte ale lui G_A, deoarece mulţimea $A \cup \{(u, v)\}$ trebuie să fie aciclică.

Bucla din liniile 2–4 ale algoritmului AAM-Generic este executată de $|V| - 1$ ori deoarece fiecare din cele $|V| - 1$ muchii ale arborelui minim de acoperire este determinată succesiv. Iniţial, când $A = \emptyset$, există $|V|$ arbori în G_A şi fiecare iteraţie reduce acest număr cu 1. Când pădurea conţine doar un singur arbore, algoritmul se termină.

Cei doi algoritmi din secţiunea 24.2 folosesc următorul corolar la terorema 24.1.

Corolarul 24.2 Fie $G = (V, E)$ un graf neorientat conex, cu o funcţie de cost w având valori reale, definită pe E. Fie A o submulţime a lui E care este inclusă într-un arbore minim de acoperire al lui G, şi fie C o componentă conexă (arbore) din pădurea $G_A = (V, A)$. Dacă (u, v) este o muchie uşoară care uneşte componenta C cu o altă componentă din G_A, atunci (u, v) este sigură pentru A.

Demonstraţie. Tăietura $(C, V - C)$ respectă mulţimea A, şi, de aceea, (u, v) este o muchie uşoară pentru această tăietură. ∎

Exerciţii

24.1-1 Fie (u, v) o muchie de cost minim dintr-un graf G. Demonstraţi că (u, v) aparţine unui arbore minim de acoperire al lui G.

24.1-2 Profesorul Sabatier formulează următoarea reciprocă a teoremei 24.1: fie $G = (V, E)$ un graf neorientat, conex cu o funcţie de cost w având valori reale, definită pe E. Fie A o submulţime a lui E care este inclusă într-un arbore minim de acoperire oarecare al lui G, fie $(S, V - S)$ orice tăietură a lui G care respectă mulţimea A şi fie (u, v) o muchie sigură pentru A care traversează $(S, V - S)$. Atunci (u, v) este o muchie uşoară pentru tăietura respectivă. Arătaţi, dând un contraexemplu, că această reciprocă nu este adevărată.

24.1-3 Arătaţi că, dacă o muchie (u, v) este conţinută într-un arbore minim de acoperire, atunci ea este o muchie uşoară care traversează o tăietură oarecare a grafului.

24.1-4 Daţi un exemplu simplu de graf, astfel încât mulţimea tuturor muchiilor uşoare care traversează o tăietură oarecare din graf nu formează un arbore minim de acoperire.

24.1-5 Fie e o muchie de cost maxim dintr-un ciclu al grafului $G = (V, E)$. Demonstraţi că există un arbore minim de acoperire al lui $G' = (V, E - \{e\})$ care este, de asemenea, un arbore minim de acoperire pentru G.

24.1-6 Arătaţi că un graf are un arbore minim de acoperire unic dacă, pentru fiecare tăietură a grafului, există o unică muchie uşoară care traversează tăietura. Arătaţi, dând un contraexemplu, că reciproca nu este adevărată.

24.1-7 Argumentaţi faptul că, dacă toate costurile muchiilor unui graf sunt pozitive, atunci orice submulţime de muchii care conectează toate vârfurile şi are un cost total minim este un arbore. Daţi un exemplu pentru a arăta că aceeaşi concluzie nu este adevărată dacă acceptăm şi costuri negative.

24.1-8 Fie T un arbore minim de acoperire al unui graf G şi fie L lista ordonată a costurilor muchiilor din T. Arătaţi că, pentru orice alt arbore minim de acoperire T' al lui G, lista L este, de asemenea, lista ordonată a costurilor muchiilor lui T'.

24.1-9 Fie T un arbore minim de acoperire al unui graf $G = (V, E)$ şi fie V' o submulţime a lui V. Fie T' subgraful lui T indus de V' şi fie G' subgraful lui G indus de V'. Arătaţi că, dacă T' este conex, atunci T' este un arbore minim de acoperire al lui G'.

24.2. Algoritmii lui Kruskal şi Prim

Cei doi algoritmi pentru determinarea unui arbore de acoperire minim descrişi în această secţiune sunt rafinări ale algoritmului generic. Fiecare foloseşte o regulă specifică pentru determinarea unei muchii sigure din linia 3 a algoritmului AAM-Generic. În algoritmul lui Kruskal, mulţimea A este o pădure. Muchia sigură adăugată la A este, întotdeauna, o muchie de cost minim din graf care uneşte două componente distincte. În algoritmul lui Prim, mulţimea A formează un singur arbore. Muchia sigură adăugată la A este, întotdeauna, o muchie de cost minim care uneşte arborele cu un vârf care nu se află în el.

Algoritmul lui Kruskal

Algoritmul lui Kruskal se bazează, direct, pe algoritmul generic pentru determinarea unui arbore minim de acoperire, prezentat în secţiunea 24.1. Algoritmul găseşte o muchie sigură, pentru a o adăuga la pădurea dezvoltată, căutând muchia (u, v) de cost minim dintre toate muchiile care unesc doi arbori din pădurea respectivă. Fie C_1 şi C_2 cei doi arbori care sunt uniţi de muchia (u, v). Deoarece (u, v) trebuie să fie o muchie uşoară care uneşte pe C_1 cu un alt arbore, corolarul 24.2 implică faptul că (u, v) este o muchie sigură pentru C_1. Algoritmul lui Kruskal este un algoritm greedy pentru că la fiecare pas adaugă pădurii o muchie cu cel mai mic cost posibil.

Această implementare a algoritmului lui Kruskal se aseamănă cu cea a algoritmului pentru calcularea componentelor conexe din secţiunea 22.1. Este folosită o structură de date pentru mulţimi disjuncte pentru reprezentarea mai multor mulţimi de elemente disjuncte. Fiecare mulţime conţine vârfurile unui arbore din pădurea curentă. Funcţia Găseşte-Mulţime(u) returnează un element reprezentativ din mulţimea care îl conţine pe u. Astfel, putem determina dacă două vârfuri u şi v aparţin aceluiaşi arbore testând dacă Găseşte-Mulţime(u) este egal cu Găseşte-Mulţime(v). Combinarea arborilor este realizată de procedura Uneşte.

AAM-Kruskal(G, w)
1: $A \leftarrow \emptyset$.
2: **pentru** fiecare vârf $v \in V[G]$ **execută**
3: Formează-Mulţime(v)
4: sortează muchiile din E în ordinea crescătoare a costului w
5: **pentru** fiecare muchie $(u, v) \in E$, în ordinea crescătoare a costului **execută**
6: **dacă** Găseşte-Mulţime(u) $\neq$Găseşte-Mulţime(v) **atunci**
7: $A \leftarrow A \cup \{(u, v)\}$
8: Uneşte(u, v)
9: **returnează** A

Modul de lucru al algoritmului lui Kruskal este prezentat în figura 24.4. Liniile 1–3 iniţializează mulţimea A cu mulţimea vidă şi creează $|V|$ arbori, unul pentru fiecare vârf. Muchiile

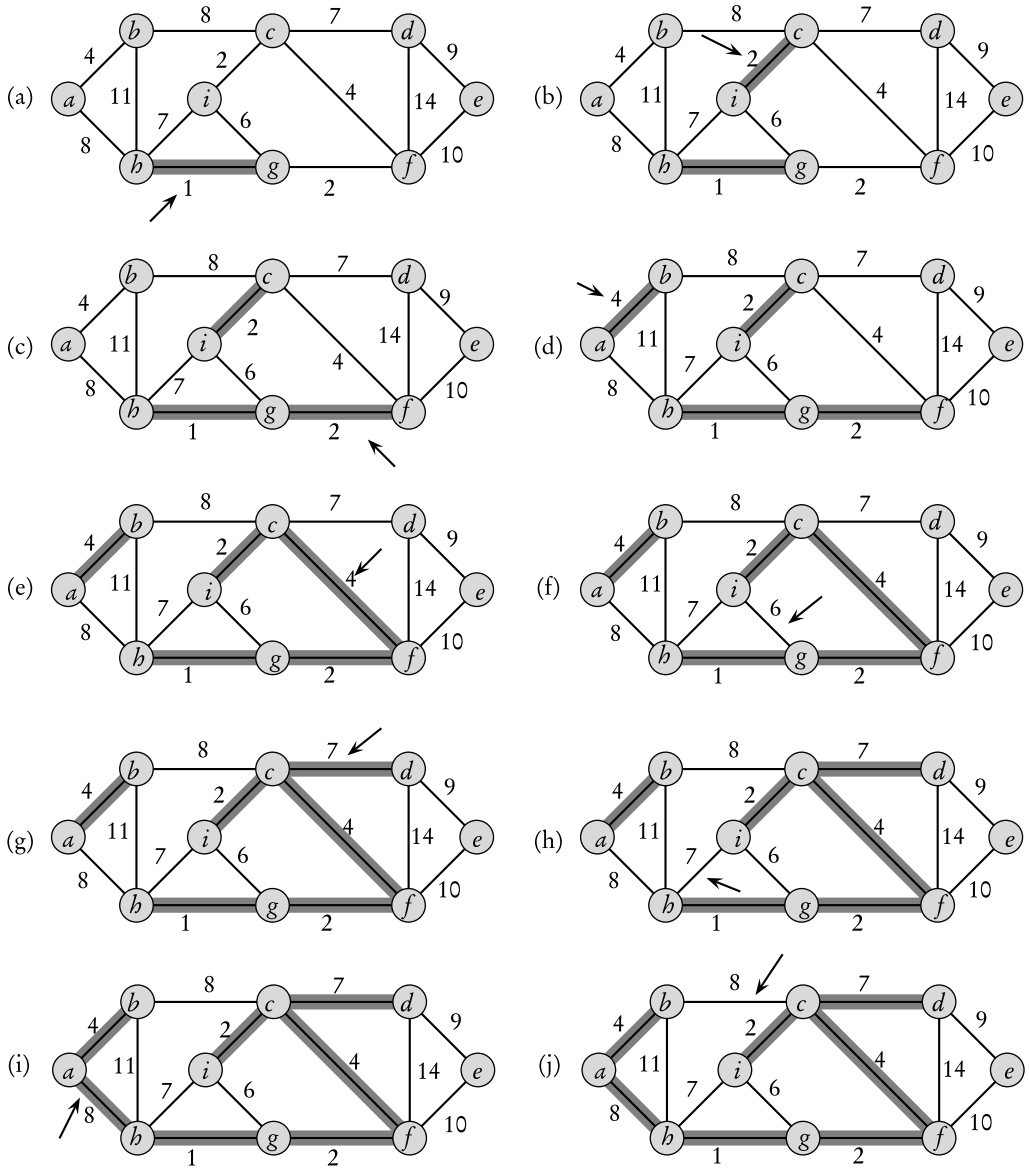
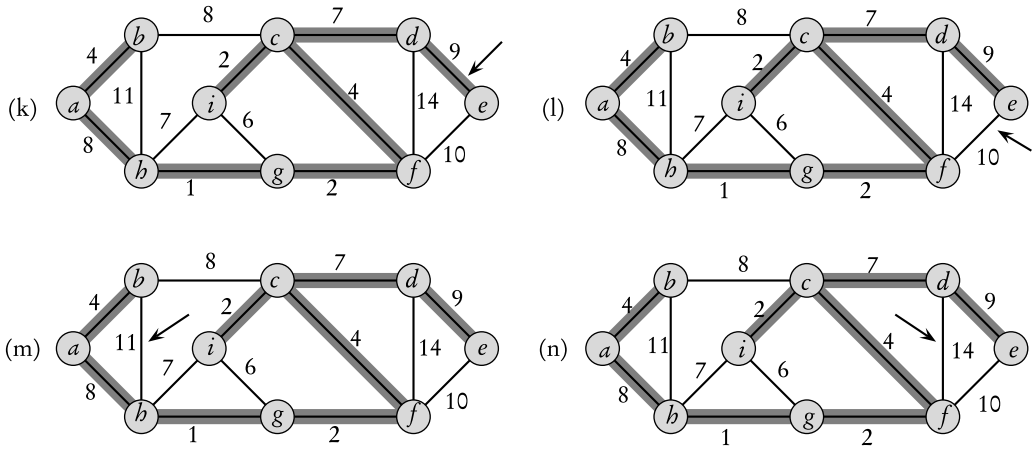


Figura 24.4 Execuţia algoritmului lui Kruskal pe graful din figura 24.1. Muchiile haşurate aparţin pădurii A care este dezvoltată. Muchiile sunt luate în considerare în ordinea crescătoare a costurilor. O săgeată arată muchia care este luată în considerare la fiecare pas al algoritmului. Dacă muchia uneşte doi arbori distincţi din pădure, ea este adăugată la pădure, unind astfel cei doi arbori.



din E sunt ordonate crescător după cost, în linia 4. Bucla **pentru** din liniile 5–8 verifică, pentru fiecare muchie (u, v), dacă punctele terminale u şi v aparţin aceluiaşi arbore. Dacă fac parte din acelaşi arbore, atunci muchia (u, v) nu poate fi adăugată la pădure fără a se forma un ciclu şi ea este abandonată. Altfel, cele două vârfuri aparţin unor arbori diferiţi, şi muchia (u, v) este adăugată la A în linia 7, vârfurile din cei doi arbori fiind reunite în linia 8.

Timpul de execuţie al algoritmului lui Kruskal pentru un graf $G = (V, E)$ depinde de implementarea structurilor de date pentru mulţimi disjuncte. Vom presupune că se foloseşte implementarea de pădure cu mulţimi distincte, din secţiunea 22.3, cu tehnicile euristice de uniune după rang şi de comprimare a drumului, deoarece este cea mai rapidă implementare cunoscută, din punct de vedere asimptotic. Iniţializarea se face într-un timp de ordinul $O(V)$, iar timpul necesar pentru sortarea muchiilor în linia 4 este de ordinul $O(E \lg E)$. Există $O(E)$ operaţii pe pădurea cu mulţimi distincte, care necesită un timp total de ordinul $O(E\ \alpha(E, V))$, unde α este inversa funcţiei lui Ackermann definită în secţiunea 22.4. Deoarece $\alpha(E, V) = O(\lg E)$, timpul total de execuţie pentru algoritmul lui Kruskal este de ordinul $O(E \lg E)$.

Algoritmul lui Prim

La fel ca algoritmul lui Kruskal, algoritmul lui Prim este un caz particular al algoritmului generic pentru determinarea unui arbore de acoperire minim prezentat în secţiunea 24.1. Modul de operare al algoritmului lui Prim se aseamănă foarte mult cu modul de operare al algoritmului lui Dijkstra pentru determinarea drumurilor de lungime minimă dintr-un graf. (Vezi secţiunea 25.2.) Algoritmul lui Prim are proprietatea că muchiile din mulţimea A formează întotdeauna un singur arbore. Conform figurii 24.5, arborele porneşte dintr-un vârf arbitrar r şi creşte până când acoperă toate vârfurile din V. La fiecare pas, se adaugă arborelui o muchie uşoară care uneşte mulţimea A cu un vârf izolat din $G_A = (V, A)$. Din corolarul 24.2, această regulă adaugă numai muchii care sunt sigure pentru A. De aceea, când algoritmul se termină, muchiile din A formează un arbore minim de acoperire. Aceasta este o strategie "greedy", deoarece arborelui îi este adăugată, la fiecare pas, o muchie care adaugă cel mai mic cost la costul total al arborelui.

Cheia implementării eficiente a algoritmului lui Prim este să procedăm în aşa fel încât să fie uşor să selectăm o nouă muchie pentru a fi adăugată la arborele format de muchiile din A. În pseudocodul de mai jos, graful conex G şi rădăcina r a arborelui minim de acoperire,

care urmează a fi dezvoltat, sunt privite ca date de intrare pentru algoritm. În timpul execuţiei algoritmului, toate vârfurile care *nu* sunt în arbore se află într-o coadă de prioritate Q bazată pe un câmp *cheie*. Pentru fiecare vârf v, $cheie[v]$ este costul minim al oricărei muchii care îl uneşte pe v cu un vârf din arbore. Prin convenţie, $cheie[v] = \infty$ dacă nu există nici o astfel de muchie. Câmpul $\pi[v]$ reţine "părintele" lui v din arbore. În timpul algoritmului, mulţimea A din algoritmul AAM-GENERIC este păstrată implicit ca

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}.$$

Când algoritmul se termină, coada de prioritate Q este vidă. Arborele minim de acoperire A al lui G este astfel

$$A = \{(v, \pi[v]) : v \in V - \{r\}\}.$$

Modul de lucru al algoritmului lui Prim este prezentat în figura 24.5. În liniile 1–4 se iniţializează coada de prioritate Q, astfel încât aceasta să conţină toate vârfurile şi se iniţializează câmpul *cheie* al fiecărui vârf cu ∞, excepţie facând rădăcina r, al cărei câmp *cheie* este iniţializat cu 0. În linia 5 se iniţializează $\pi[r]$ cu NIL, deoarece rădăcina r nu are nici un părinte. Pe parcursul algoritmului, mulţimea $V - Q$ conţine vârfurile arborelui curent. În linia 7 este identificat un vârf $u \in Q$ incident unei muchii uşoare care traversează tăietura $(V - Q, Q)$ (cu excepţia primei iteraţii, în care $u = r$ datorită liniei 4). Eliminarea lui u din mulţimea Q îl adaugă pe acesta mulţimii $V - Q$ a vârfurilor din arbore. În liniile 8–11 se actualizează câmpurile *cheie* şi π ale fiecărui vârf v adiacent lui u, dar care nu se află în arbore. Actualizarea respectă condiţiile $cheie[v] = w(v, \pi[v])$, şi $(v, \pi[v])$ să fie o muchie uşoară care îl uneşte pe v cu un vârf din arbore.

```
AAM-PRIM(G, w, r)
 1: Q ← V[G]
 2: pentru fiecare u ∈ Q execută
 3:    cheie[u] ← ∞
 4: cheie[r] ← 0
 5: π[r] ←NIL
 6: cât timp Q ≠ ∅ execută
 7:    u ←EXTRAGE-MIN(Q)
 8:    pentru fiecare v ∈ Adj[u] execută
 9:       dacă v ∈ Q şi w(u, v) < cheie[v] atunci
10:          π[v] ← u
11:          cheie[v] ← w(u, v)
```

Performanţa algoritmului lui Prim depinde de modul în care implementăm coada de prioritate Q. Dacă Q este implementată folosind heap-uri binare (vezi capitolul 7), putem folosi procedura CONSTRUIEŞTE-HEAP pentru a executa iniţializarea din liniile 1–4 într-un timp de ordinul $O(V)$. Bucla este executată de $|V|$ ori şi, deoarece fiecare procedură EXTRAGE-MIN are nevoie de un timp de ordinul $O(\lg V)$, timpul total pentru toate apelurile către EXTRAGE-MIN este $O(V \lg V)$. Bucla **pentru** din liniile 8–11 este executată în total de $O(E)$ ori, deoarece suma tuturor listelor de adiacenţă este $2|E|$. În cadrul buclei **pentru**, testul pentru apartenenţa la Q din linia 9 poate fi implementat într-un timp constant, păstrând un bit pentru fiecare vârf, care ne spune dacă vârful respectiv se află în Q sau nu, şi reactualizând bitul când vârful este eliminat din Q.

Atribuirea din linia 11 implică o procedură explicită MICŞOREAZĂ-CHEIE asupra heap-ului, care poate fi implementată în cadrul unui heap binar într-un timp de ordinul $O(\lg V)$. Astfel,

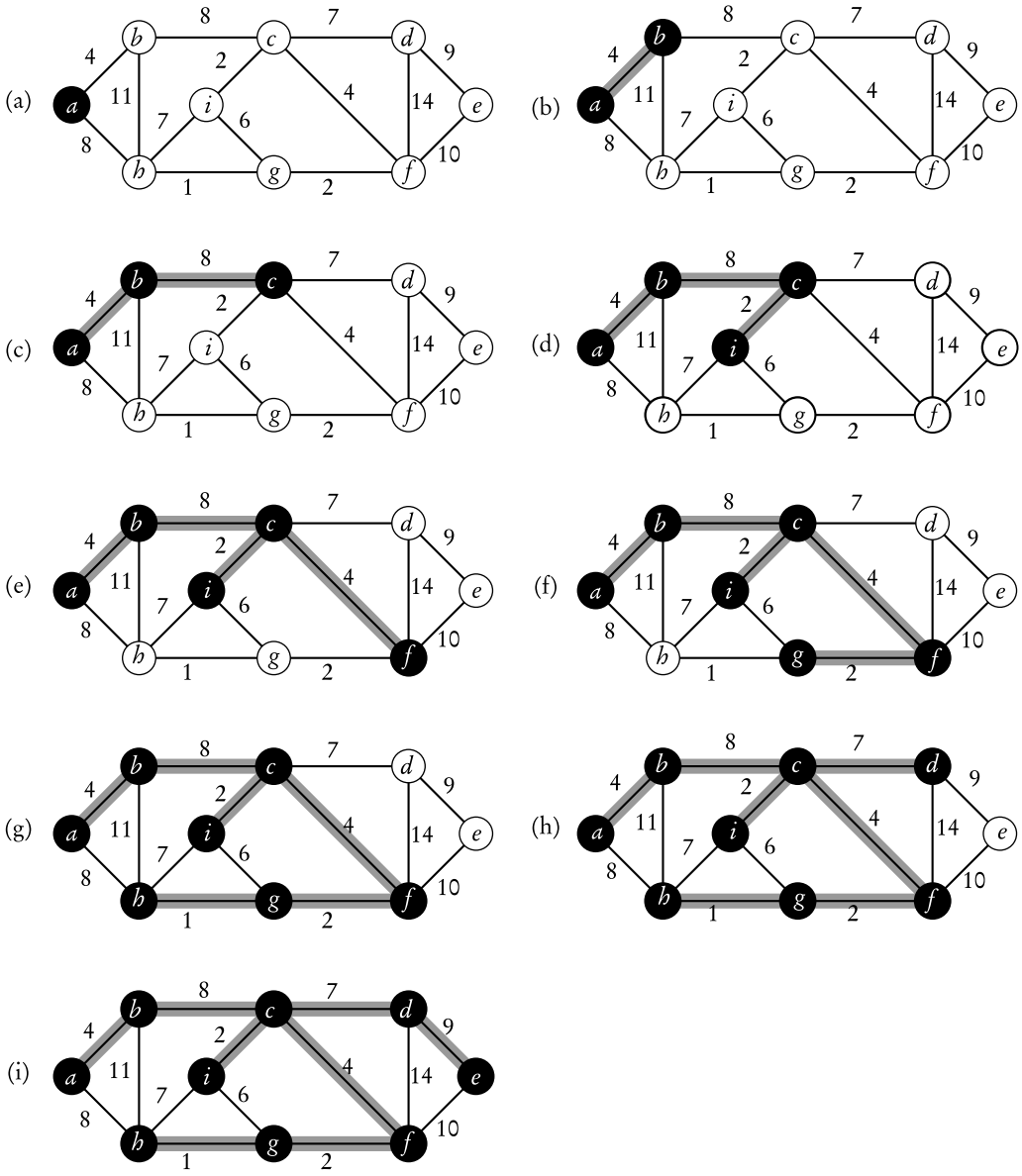


Figura 24.5 Execuţia algoritmului lui Prim pe graful din figura 24.1. Vârful rădăcină este a. Muchiile haşurate cu gri fac parte din arborele care este dezvoltat, iar vârfurile din arbore au culoarea neagră. La fiecare pas al algoritmului, vârfurile din arbore determină o tăietură a grafului, şi o muchie uşoară care traversează tăietura este adăugată arborelui. În pasul al doilea, de exemplu, algoritmul are de ales între a adăuga arborelui fie muchia (b, c), fie muchia (a, h), deoarece ambele sunt muchii uşoare care traversează tăietura.

timpul total necesar pentru algoritmul lui Prim este $O(V \lg V + E \lg V) = O(E \lg V)$, care este acelaşi, din punct de vedere asimptotic, cu cel al algoritmului lui Kruskal.

Totuşi, timpul asimptotic de execuţie al algoritmului lui Prim poate fi îmbunătăţit, folosind heap-uri Fibonacci. În capitolul 21 se arată că dacă $|V|$ elemente sunt organizate în heap-uri Fibonacci, putem efectua o operaţie EXTRAGE-MIN într-un timp amortizat $O(\lg V)$ şi o operaţie MICŞOREAZĂ-CHEIE (pentru a implementa linia 11) într-un timp amortizat $O(1)$. De aceea, dacă folosim un ansambu Fibonacci pentru a implementa coada de prioritate Q, timpul de rulare al algoritmului lui Prim se îmbunătăţeşte ajungând la $O(E + V \lg V)$.

Exerciţii

24.2-1 Algoritmul lui Kruskal poate returna arbori minimi de acoperire diferiţi pentru acelaşi graf de intrare G, în funcţie de modul în care muchiile având costuri egale sunt aranjate atunci când ele sunt ordonate. Arătaţi că, pentru fiecare arbore minim de acoperire T al unui graf G, există o ordonare a muchiilor lui G, astfel încât algoritmul lui Kruskal returnează T.

24.2-2 Să presupunem că graful $G = (V, E)$ este reprezentat printr-o matrice de adiacenţă. Daţi o implementare simplă a algoritmului lui Prim pentru acest caz care rulează în timpul $O(V^2)$.

24.2-3 Este implementarea folosind heap-uri Fibonacci mai rapidă asimptotic decât reprezentarea folosind heap-uri binare pentru un graf rar $G = (V, E)$, unde $|E| = \Theta(V)$? Dar în cazul unui graf dens, unde $|E| = \Theta(V^2)$? În ce relaţie trebuie să fie $|E|$ şi $|V|$ pentru ca implementarea folosind heap-uri Fibonacci să fie mai rapidă decât implementarea folosind heap-uri binare?

24.2-4 Să presupunem că toate muchiile dintr-un graf au costurile reprezentate prin numere întregi cuprinse între 1 şi $|V|$. Cât de rapid poate deveni algoritmul lui Kruskal? Dar în cazul în care costurile muchiilor sunt întregi între 1 şi W, unde W este o constantă dată?

24.2-5 Presupuneţi că toate muchiile dintr-un graf au costurile reprezentate prin numere întregi cuprinse între 1 şi $|V|$. Cât de rapid poate deveni algoritmul lui Prim? Dar în cazul în care costurile muchiilor sunt întregi între 1 şi W, unde W este o constantă dată?

24.2-6 Descrieţi un algoritm eficient care, dat fiind un graf neorientat G, determină un arbore de acoperire al lui G în care costul maxim al unei muchii este minim dintre toţi arborii de acoperire ai lui G.

24.2-7 $\star$ Să presupunem că într-un graf costurile muchiilor sunt uniform distribuite de-a lungul intervalului semideschis $[0, 1)$. Care algoritm poate deveni mai rapid, al lui Kruskal sau al lui Prim?

24.2-8 $\star$ Să presupunem că un graf G are un arbore minim de acoperire deja calculat. Cât de rapid poate fi actualizat arborele minim de acoperire dacă lui G i se adaugă un nou vârf şi muchii incidente?

Probleme

24-1 Al doilea arbore de acoperire minim

Fie $G = (V, E)$ un graf conex, neorientat cu funcţia de cost $w : E \to \mathbb{R}$, şi presupuneţi că $|E| \geq |V|$.

a. Fie T un arbore minim de acoperire al lui G. Demonstraţi că există muchiile $(u, v) \in T$ şi $(x, y) \notin T$, astfel încât $T - \{(u, v)\} \cup \{(x, y)\}\}$ este un al doilea arbore minim de acoperire al lui G.

b. Fie T un arbore de acoperire al lui G şi, pentru oricare două vârfuri $u, v \in V$, fie $max[u, v]$ muchia de cost maxim pe drumul unic de la u la v din arborele T. Descrieţi un algoritm care să ruleze într-un timp de ordinul $O(V^2)$ care, dat fiind T, calculează $max[u, v]$ pentru toate vârfurile $u, v \in V$.

c. Elaboraţi algoritmul care calculează cel de-al doilea arbore de acoperire minim al lui G.

24-2 Arbore de acoperire minim în grafuri rare

Pentru un graf conex foarte rar $G = (V, E)$, putem îmbunătăţi timpul de execuţie $O(E + V \lg V)$ al algoritmului lui Prim cu heap-uri Fibonacci "preprocesând" graful G pentru a micşora numărul de vârfuri înaintea rulării algoritmului lui Prim.

AAM-REDUCERE(G, T)
1: **pentru** fiecare $v \in V[G]$ **execută**
2: $marcaj[v] \leftarrow$ FALS
3: CONSTRUIEŞTE-MULŢIME(v)
4: **pentru** fiecare $u \in V[G]$ **execută**
5: **dacă** $marcaj[u] =$FALS **atunci**
6: alege $v \in Adj[u]$ astfel încât $w[u, v]$ este minimizat
7: UNIRE(u, v)
8: $T \leftarrow T \cup \{orig[u, v]\}$
9: $marcaj[u] \leftarrow marcaj[v] \leftarrow$ ADEVĂRAT
10: $V[G'] \leftarrow \{$GĂSEŞTE-MULŢIME$(v) : v \in V[G]\}$
11: $E[G'] \leftarrow \emptyset$
12: **pentru** fiecare $(x, y) \in E[G]$ **execută**
13: $u \leftarrow$GĂSEŞTE-MULŢIME(x)
14: $v \leftarrow$GĂSEŞTE-MULŢIME(y)
15: **dacă** $(u, v) \notin E[G']$ **atunci**
16: $E[G'] \leftarrow E[G'] \cup \{(u, v)\}$
17: $orig[u, v] \leftarrow orig[x, y]$
18: $w[u, v] \leftarrow w[x, y]$
19: **altfel dacă** $w[x, y] < w[u, v]$ **atunci**
20: $orig[u, v] \leftarrow orig[x, y]$
21: $w[u, v] \leftarrow w[x, y]$
22: construieşte listele de adiacenţă Adj pentru G'
23: **returnează** G' şi T

Procedura AAM-REDUCERE primeşte ca argument de intrare un graf cu costuri G şi returnează o versiune "contractată" a lui G, având adăugate câteva muchii la arborele minim de acoperire T curent. Iniţial, pentru fiecare muchie $(u, v) \in E$, presupunem că $orig[u, v] = (u, v)$ şi că $w[u, v]$ este costul muchiei.

a. Fie T mulţimea de muchii returnată de AAM-REDUCERE şi fie T' un arbore minim de acoperire G' returnat de procedură. Demonstraţi că $T \cup \{orig[x, y] : (x, y) \in T'\}$ este un arbore minim de acoperire al lui G.

b. Argumentaţi că $|V[G']| \leq |V|/2$.

c. Arătaţi cum se poate implementa AAM-REDUCERE, astfel încât să ruleze în timpul $O(E)$. (*Indicaţie:* Folosiţi structuri de date simple.)

d. Să presupunem că rulăm k faze ale AAM-REDUCERE, folosind graful produs de una din faze ca intrare pentru următoarea şi acumulând muchiile în T. Argumentaţi faptul că timpul total de rulare al celor k faze este $O(kE)$.

e. Să presupunem că, după ce rulăm k faze ale algoritmului AAM-REDUCERE, rulăm algoritmul lui Prim pe graful produs de ultima fază. Arătaţi cum trebuie ales k pentru ca timpul total de execuţie să fie $O(E \lg \lg V)$. Argumentaţi că alegerea lui k minimizează timpul de rulare asimptotic total.

f. Pentru ce valori ale lui $|E|$ (relativ la $|V|$) este mai rapid asimptotic algoritmul lui Prim cu preprocesare decât algoritmul lui Prim fără preprocesare?

Note bibliografice

Tarjan [188] analizează problema arborelui de acoperire minim şi oferă un material avansat excelent. Un istoric al problemei arborelui de acoperire minim a fost scris de către Graham şi Hell [92].

Tarjan atribuie prima problemă a arborelui minim de acoperire unei lucrări din 1926 a lui O. Borůvka. Algoritmul lui Kruskal a fost anunţat de către Kruskal [131] în 1956. Algoritmul cunoscut, de obicei, sub numele de algoritmul lui Prim a fost într-adevăr inventat de către Prim [163], dar a fost, de asemenea, inventat mai înainte de către V. Jarník în 1930.

Motivul pentru care algoritmii greedy sunt efectivi în determinarea arborilor minimi de acoperire este că mulţimea arborilor unui graf formează un matroid grafic. (Vezi secţiunea 17.4.)

Cel mai rapid algoritm pentru determinarea unui arbore minim de acoperire în cazul când $|E| = \Omega(V \lg V)$ este algoritmul lui Prim implementat cu heap-uri Fibonacci. Pentru grafuri mai rare, Fredman şi Tarjan [75] dau un algoritm care rulează în timpul $O(E \lg \beta(|E|, |V|))$, unde $\beta(|E|, |V|) = \min\{i : \lg^{(i)} |V| \leq |E|/|V|\}$. Faptul că $|E| \geq |V|$ implică faptul că algoritmul lor rulează în timpul $O(E \lg^{\star} V)$.

25 Drumuri minime de sursă unică

Să presupunem că un automobilist doreşte să găsească cel mai scurt drum de la Chicago la Boston, utilizând o hartă rutieră a Statelor Unite pe care sunt indicate distanţele între fiecare două intersecţii adiacente.

Un mod posibil de rezolvare a acestei probleme este de a enumera toate drumurile de la Chicago la Boston şi, pe baza lungimilor drumurilor, de a selecta unul din cele de lungime minimă. Este uşor de văzut, chiar şi în cazul în care sunt considerate numai drumurile care nu conţin cicluri, că numărul variantelor este enorm, şi în plus, majoritatea lor nu pot fi luate în considerare. De exemplu, un drum de la Chicago la Boston via Houston este evident o soluţie de neacceptat, deoarece Houston se află la distanţa de 1000 de mile de oraşele considerate.

În acest capitol şi în capitolul 26, vom arăta cum poate fi rezolvată această problemă în mod eficient. Într-o ***problemă de drum minim***, este dat un graf orientat cu costuri $G = (V, E)$, funcţia de cost $w : E \to \mathbb{R}$ asociind fiecărei muchii câte un cost exprimat printr-un număr real. ***Costul*** unui drum $p = \langle v_0, v_1, ..., v_k \rangle$ este suma costurilor corespunzătoare muchiilor componente:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Costul unui drum minim (costul optim) de la u la v se defineşte prin:

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{dacă există drum de la } u \text{ la } v, \\ \infty & \text{altfel.} \end{cases}$$

Un ***drum minim*** de la u la v este orice drum p cu proprietatea $w(p) = \delta(u, v)$.

În cazul exemplului Chicago-Boston, putem modela harta rutieră printr-un graf: vârfurile reprezintă punctele de intersecţie, muchiile şi costurile acestora reprezintă segmentele de drum, respectiv distanţele între intersecţii. Scopul pe care ni-l propunem este de a găsi drumul minim între o intersecţie din Chicago (de exemplu Clark Street cu Addison Avenue) şi o intersecţie dată din Boston (Brookline Avenue cu Yawkey Way).

Costurile muchiilor trebuie interpretate mai curând ca metrice decât ca distanţe, ele fiind utilizate frecvent pentru a reprezenta timpul, costul, penalităţi sau alte mărimi care se acumulează liniar de-a lungul unui drum şi pe care, de regulă, dorim să le minimizăm.

Algoritmul de căutare în lăţime prezentat în secţiunea 23.2 este un algoritm pentru determinarea drumurilor minime în grafuri fără costuri, adică, grafuri ale căror muchii sunt considerate având costul egal cu unitatea. Deoarece multe dintre conceptele utilizate în cadrul descrierii algoritmului de căutare în lăţime vor fi considerate şi în studiul drumurilor minime în grafuri cu costuri, sfătuim cititorul ca înainte de a continua lectura acestui capitol, să revada secţiunea 23.3.

Variante

În acest capitol ne vom concentra pe ***problema drumurilor minime de sursă unică***: fiind dat un graf $G = (V, E)$, dorim să găsim pentru un vârf ***sursă*** $s \in V$ dat, câte un drum

minim de la s la fiecare vârf $v \in V$. Pe baza unui algoritm care rezolvă această problemă, pot fi rezolvate o serie de probleme reductibile la aceasta, dintre care menţionăm următoarele variante.

Problema drumurilor minime de destinaţie unică: Să se determine pentru fiecare vârf $v \in V$, câte un drum minim de la v la un vârf ***destinaţie*** t prestabilit. Inversând orientarea fiecărei muchii a grafului, problema se reduce la o problemă de determinare a drumurilor minime de sursă unică.

Problema drumurilor minime de sursă şi destinaţie unică: Să se determine un drum minim de la u la v pentru u şi v date. Dacă rezolvăm problema drumurilor minime de sursă unică pentru vârful u atunci rezolvăm şi această problemă. Mai mult decât atât, nu se cunosc algoritmi pentru rezolvarea acestei probleme care să fie asimptotic mai rapizi decât cel mai bun algoritm pentru determinarea drumurilor minime de sursă unică, în cazul cel mai defavorabil.

Problema drumurilor minime pentru surse şi destinaţii multiple: Pentru fiecare pereche de vârfuri u şi v, să se determine câte un drum minim de la u la v. Problema poate fi rezolvată, de exemplu, prin aplicarea algoritmului pentru sursă unică, pentru fiecare vârf al grafului. În mod obişnuit, această problemă poate fi rezolvată mai eficient şi structura ei este interesantă prin ea însăşi. În cadrul capitolului 26, problema drumurilor minime între perechile de vârfuri ale unui graf va fi reluată şi analizată în detaliu.

Muchii de cost negativ

În cazul algoritmului pentru determinarea drumurilor minime de sursă unică, anumite muchii ale grafului pot avea asociate costuri negative. Dacă graful $G = (V, E)$ nu conţine cicluri accesibile din vârful sursă s şi care să fie de cost negativ, atunci pentru orice $v \in V$, costul optim $\delta(s, v)$ este bine definit, chiar dacă este de valoare negativă. Dacă însă există cel puţin un ciclu de cost negativ accesibil din s, costurile optimale nu sunt bine definite. Într-adevăr nici un drum de la s la un vârf component al ciclului nu poate fi un drum minim deoarece întotdeauna se poate construi un drum "mai scurt" şi anume de-a lungul drumului considerat urmat de o traversare a ciclului de cost negativ. În cazul în care există cel puţin un drum de la s la v care să conţină un ciclu de cost negativ, prin definiţie $\delta(s, v) = -\infty$.

În figura 25.1 este ilustrat efectul costurilor negative asupra costurilor drumurilor minime. Deoarece există un singur drum de la s la a (şi anume drumul $\langle s, a\rangle$), $\delta(s, a) = w(s, a) = 3$. De asemenea, există un singur drum de la s la b şi deci $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. Evident, există o infinitate de drumuri de la s la c: $\langle s, c\rangle, \langle s, c, d, c\rangle, \langle s, c, d, c, d, c\rangle$ etc. Deoarece ciclul $\langle c, d, c\rangle$ are costul $6 + (-3) = 3 > 0$, drumul minim de la s la c este $\langle s, c\rangle$ de cost $\delta(s, c) = 5$. De asemenea, drumul minim de la s la d este $\langle s, c, d\rangle$ de cost $\delta(s, d) = w(s, c) + w(c, d) = 11$. În mod analog, există o infinitate de drumuri de la s la e: $\langle s, e\rangle, \langle s, e, f, e\rangle, \langle s, e, f, e, f, e\rangle$ etc. Evident ciclul $\langle e, f, e\rangle$ este de cost $3 + (-6) = -3 < 0$ dar nu există un drum minim de la s la e deoarece prin traversarea repetată a ciclului de cost negativ $\langle e, f, e\rangle$ rezultă drumuri de la s la e de costuri negative oricât de mari şi deci $\delta(s, e) = -\infty$. Acelaşi argument justifică $\delta(s, f) = -\infty$. Deoarece g este accesibil din f, există drumuri de la s la g de costuri negative oricât de mari, deci $\delta(s, g) = -\infty$. Vârfurile h, i şi j compun de asemenea un ciclu având cost negativ. Deoarece încă nici unul dintre aceste vârfuri nu este accesibil din s, rezultă $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

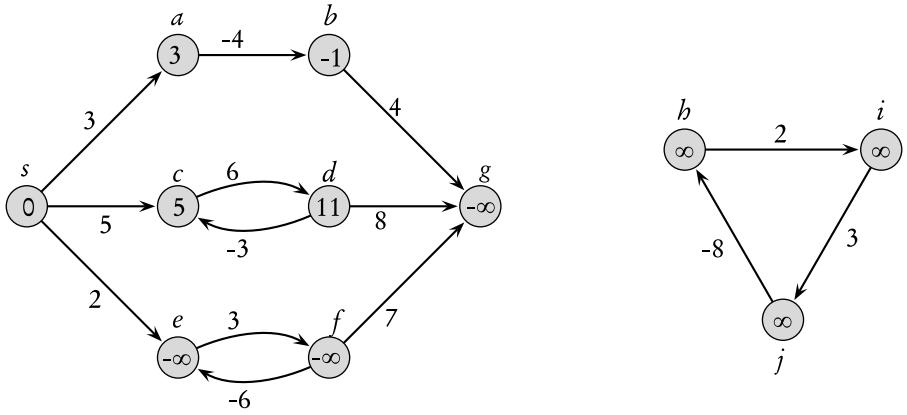


Figura 25.1 Costuri negative într-un graf orientat. Pe fiecare vârf este scris costul drumului minim dintre acesta şi sursa s. Deoarece vârfurile e şi f formează un ciclu de cost negativ accesibil din s, ele au costurile pentru drumul minim egale cu $-\infty$. Deoarece vârful g este accesibil dintr-un vârf pentru care costul drumului minim este $-\infty$, şi acesta are costul drumului minim egal cu $-\infty$. Vârfuri ca h, i, şi j nu sunt accesibile din s, şi de aceea costurile lor pentru drumul minim sunt egale cu ∞, deşi fac parte dintr-un ciclu având cost negativ.

O serie de algoritmi pentru determinarea drumurilor, cum este de exemplu algoritmul Dijkstra, impun ca toate costurile muchiilor grafului de intrare să fie nenegative, ca în cazul exemplului cu harta rutieră. Alţi algoritmi, cum este de exemplu algoritmul Bellman-Ford, acceptă ca graful de intrare să aibă muchii cu costuri negative şi produc răspunsuri corecte în cazul în care nici un ciclu de cost negativ nu este accesibil din vârful sursă considerat. În mod obişnuit, aceşti algoritmi detectează şi semnalează prezenţa ciclurilor de costuri negative.

Reprezentarea drumurilor minime

Frecvent, dorim să determinăm nu numai costurile optimale, dar şi vârfurile care compun un drum minim. Pentru reprezentarea unui drum minim vom utiliza o reprezentare similară celei considerate în cazul arborilor de lăţime în secţiunea 23.2. Fiind dat un graf $G = (V, E)$, se reţine pentru fiecare vârf $v \in V$ un ***predecesor*** $\pi[v]$ care este fie un alt vârf, fie NIL. Algoritmii pentru determinarea drumurilor minime prezentaţi în cadrul acestui capitol determină π astfel încât pentru orice vârf v, lanţul de predecesori care începe cu v să corespundă unei traversări în ordine inversă a unui drum minim de la s la v. Astfel, pentru orice vârf v, pentru care $\pi[v] \neq$ NIL, procedura TIPĂREŞTE-DRUM(G, s, v) din secţiunea 23.2 poate fi utilizată pentru tipărirea unui drum minim de la s la v.

Pe durata execuţiei unui algoritm pentru determinarea drumurilor minime, valorile lui π nu indică în mod necesar drumurile minime. Ca şi în cazul căutării în lăţime, vom considera ***subgraful predecesor*** $G_\pi = (V_\pi, E_\pi)$ indus de valorile lui π, unde V_π este mulţimea vârfurilor din G având proprietatea că au predecesor diferit de NIL, reunită cu mulţimea constând din vârful sursă s:

$$V_\pi = \{v \in V : \pi[v] \neq NIL\} \cup \{s\}$$

Mulţimea de muchii orientate E_π este mulţimea de muchii indusă de valorile lui π pentru vârfurile din V_π:

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi \setminus \{s\}\}$$

Vom demonstra că valorile π determinate de algoritmii ce vor fi prezentaţi în acest capitol asigură ca la terminare, G_π să fie un "arbore al drumurilor minime" – în sensul că este un arbore cu rădăcină conţinând câte un drum minim de la sursa s la fiecare vârf al grafului care este accesibil din s. Un arbore al drumurilor minime este similar arborelui de lăţime din secţiunea 23.2, cu diferenţa că el conţine câte un drum minim calculat în termenii costurilor muchiilor şi nu a numărului de muchii. Mai exact, fie $G = (V, E)$ un graf orientat cu costuri, având funcţia de cost $w : E \to \mathbb{R}$; presupunem că G nu conţine cicluri de cost negativ accesibile din vârful sursă s, deci drumurile minime sunt bine definite. Un ***arbore al drumurilor minime*** de rădacină s este un subgraf orientat $G' = (V', E')$, unde $V' \subseteq V$, $E' \subseteq E$ astfel încât următoarele condiţii sunt îndeplinite:

1. V' este mulţimea vârfurilor accesibile din s în G,
2. G' este un arbore orientat cu rădăcină, având rădăcina s,
3. pentru orice $v \in V'$ unicul drum de la s la v în G' este un drum minim de la s la v în G.

Drumurile minime nu sunt în mod necesar unice şi în consecinţă există mai mulţi arbori de drumuri minime. De exemplu, în figura 25.2 este prezentat un graf orientat cu costuri şi doi arbori de drumuri minime având aceeaşi rădăcină.

Prezentarea generală a capitolului

Algoritmii pentru determinarea drumurilor minime de sursă unică prezentaţi în cadrul acestui capitol sunt bazaţi pe o tehnică cunoscută sub numele de relaxare. În cadrul secţiunii 25.1 sunt demonstrate o serie de proprietăţi generale ale algoritmilor de determinare a drumurilor minime şi sunt stabilite concluzii relativ la algoritmii bazaţi pe relaxare. Algoritmul Dijkstra pentru rezolvarea problemei drumurilor minime de sursă unică, în cazul în care toate muchiile au costuri nenegative, este prezentat în secţiunea 25.2. Secţiunea 25.3 prezintă algoritmul Bellman-Ford utilizat în cazul mai general în care muchiile pot avea şi costuri negative. Algoritmul Bellman-Ford permite detectarea existenţei unui ciclu de cost negativ, accesibil din vârful sursă. În cadrul secţiunii 25.4 este prezentat un algoritm de timp liniar pentru determinarea drumurilor minime de sursă unică într-un graf aciclic orientat. În secţiunea 25.5 este demonstrat modul în care algoritmul Bellman-Ford poate fi utilizat pentru rezolvarea unui caz special de "programare liniară".

Analiza noastră impune adoptarea unor convenţii de calcul în care unul sau ambii operanzi sunt $\pm\infty$. Vom presupune că pentru orice număr real $a \neq -\infty$, avem $a + \infty = \infty + a = \infty$. De asemenea, pentru ca argumentele utilizate în demonstraţii să fie valabile şi în cazul existenţei ciclurilor de cost negativ, presupunem că pentru orice număr real $a \neq \infty$, avem $a + (-\infty) = (-\infty) + a = (-\infty)$.

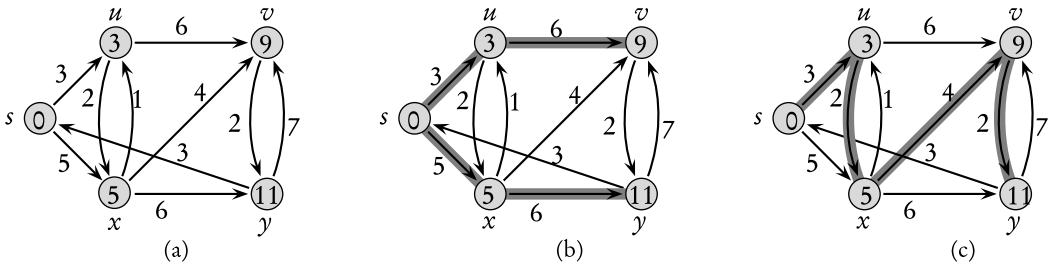


Figura 25.2 **(a)** Un graf orientat cu costuri împreună cu costurile drumurilor minime de la sursa s. **(b)** Muchiile haşurate formează un arbore de drumuri minime având drept rădăcină sursa s. **(c)** Un alt arbore de drumuri minime având aceeaşi rădăcină.

25.1. Drumuri minime şi relaxare

Pentru înţelegerea algoritmilor de determinare a drumurilor minime de sursă unică, este utilă înţelegerea tehnicilor pe care aceştia le utilizează şi a proprietăţile drumurilor minime care sunt exploatate de aceste tehnici. Tehnica principală utilizată de algoritmii descrişi în cadrul acestui capitol este relaxarea, metodă care revine la decrementarea repetată a unei margini superioare pentru determinarea costului corespunzător unui drum minim pentru fiecare vârf, până când marginea superioară devine egală cu costul unui drum optimal. În cadrul acestei secţiuni vom analiza modul în care operează relaxarea şi vom demonstra formal mai multe proprietăţi pe care aceasta le conservă.

Apreciem că la prima lectură a acestui capitol demonstraţiile teoremelor pot fi eventual omise – este suficientă reţinerea doar a enunţurilor – şi în continuare să fie abordaţi algoritmii prezentaţi în secţiunea 25.2 şi 25.3. Atragem atenţia însă asupra importanţei rezultatului stabilit de lema 25.7, deoarece acesta este esenţial în înţelegerea algoritmilor pentru determinarea drumurilor minime prezentaţi în cadrul acestui capitol. De asemenea, sugerăm cititorului că la prima lectură pot fi ignorate lemele referitoare la subgraful predecesor şi arborii de drumuri minime (lemele 25.8 şi 25.9); în schimb ar trebui să-şi concentreze atenţia asupra lemelor care se referă la costurile drumurilor minime.

Structura optimală a unui drum minim

De regulă, algoritmii pentru determinarea drumurilor minime exploatează proprietatea că un drum minim între două vârfuri conţine alte subdrumuri optimale. Această proprietate de optimalitate substructurală este o caracteristică atât a programării dinamice (capitolul 16) cât şi a metodei greedy (capitolul 17). În fapt, algoritmul Dijkstra este un algoritm greedy, iar algoritmul Floyd-Warshall pentru determinarea drumurilor minime pentru toate perechile de vârfuri (vezi capitolul 26) este un algoritm de programare dinamică. Următoarea lemă şi corolarul ei stabilesc mai precis proprietatea de optimalitate substructurală a drumurilor minime.

Lema 25.1 (Subdrumurile unui drum minim sunt drumuri optimale) Fiind dat un graf orientat cu costuri $G = (V, E)$ cu funcţia de cost $w : E \to \mathbb{R}$, fie $p = \langle v_1, v_2, ..., v_k \rangle$ un drum minim de la vârful $v_1$la vârful v_k şi pentru orice i şi j astfel încât $1 \leq i \leq j \leq k$ fie

$p_{ij} = \langle v_i, v_{i+1}, ..., v_j \rangle$ subdrumul lui p de la vârful v_i la vârful v_j. Atunci, p_{ij} este un drum minim de la v_i la v_j.

Demonstraţie. Dacă descompunem drumul p în $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, atunci $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Presupunem acum că există un drum p'_{ij} de la v_i la v_j de cost $w(p'_{ij}) < w(p_{ij})$. Atunci, $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ este un drum de la v_1 la v_k al cărui cost $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ este strict mai mic decât $w(p)$ ceea ce contrazice presupunerea că p este un drum minim de la v_1 la v_k. ∎

În cadrul căutării în lăţime (secţiunea 23.2), rezultatul stabilit de lema 23.1 este o proprietate simplă a celor mai scurte distanţe în grafuri fără costuri. Corolarul următor al lemei 25.1 generalizează această proprietate în cazul grafurilor cu costuri.

Corolarul 25.2 Fie $G = (V, E)$ un graf orientat cu costuri cu funcţia de cost $w : E \to \mathbb{R}$. Presupunem că un drum minim p de la sursa s la vârful v poate fi descompus în $s \overset{p'}{\rightsquigarrow} u \to v$ pentru un anume vârf u şi un drum p'. Atunci, costul unui drum minim de la s la v este $\delta(s, v) = \delta(s, u) + w(u, v)$.

Demonstraţie. Conform lemei 25.1, subdrumul p' este un drum minim de la sursa s la vârful u. Rezultă

$$\delta(s, v) = w(p) = w(p') + w(u, v) = \delta(s, u) + w(u, v).$$

∎

Următoarea lemă stabileşte o proprietate simplă, dar utilă, a costurilor drumurilor minime.

Lema 25.3 Fie $G = (V, E)$ un graf orientat cu costuri cu funcţia de cost $w : E \to \mathbb{R}$ şi vârful sursă s. Atunci, pentru toate muchiile $(u, v) \in E$ avem $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Demonstraţie. Un drum minim p de la sursa s la vârful v are costul mai mic sau egal cu costul oricărui alt drum de la s la v. În particular, costul drumului p este mai mic sau egal cu costul oricărui drum de la sursa s la vârful u şi care conţine muchia (u, v). ∎

Relaxare

Algoritmii prezentaţi în cadrul acestui capitol utilizează tehnica de ***relaxare***. Pentru fiecare vârf $v \in V$ păstrăm un atribut $d[v]$, reprezentând o margine superioară a costului unui drum minim de la sursa s la v. Numim $d[v]$ o ***estimare a drumului minim***. Estimările drumurilor minime şi predecesorii sunt iniţializaţi prin următoarea procedură:

INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s)
1: **pentru** fiecare vârf $v \in V[G]$ **execută**
2: $\quad d[v] \leftarrow \infty$
3: $\quad \pi[v] \leftarrow$ NIL
4: $d[s] \leftarrow 0$

După iniţializare $\pi[v] =$NIL pentru orice $v \in V, d[v] = 0$ pentru $v = s$ şi $d[v] = \infty$ pentru $v \in V - \{s\}$.

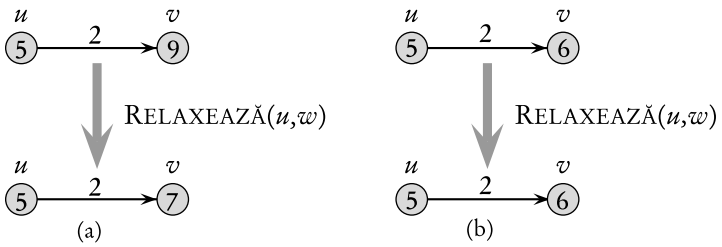


Figura 25.3 Relaxarea unei muchii (u, v) cu costul $w(u, v) = 2$. Împreună cu fiecare vârf este prezentată şi estimarea pentru drumul minim. **(a)** Deoarece, înainte de relaxare, $d[v] > d[u] + w(u, v)$, valoarea lui $d[v]$ descreşte. **(b)** Aici, înainte de pasul de relaxare, $d[v] \leq d[u] + w(u, v)$, şi deci valoarea lui $d[v]$ nu este modificată de relaxare

În procesul de ***relaxare***[1] a unei muchii (u, v) se verifică dacă drumul minim la v, determinat până la acel moment, poate fi îmbunătăţit pe baza vârfului u şi, dacă da, atunci se reactualizează $d[v]$ şi $\pi[v]$. Un pas de relaxare poate determina descreşterea valorii estimării drumului minim $d[v]$ şi reactualizarea câmpului $\pi[v]$ ce conţine predecesorul vârfului v. Pseudocodul următor realizează un pas de relaxare pe muchia (u, v).

RELAXEAZĂ(u, v, w)
1: **dacă** $d[v] > d[u]+w(u, v)$ **atunci**
2: $\quad d[v] \leftarrow d[u] + w(u, v)$
3: $\quad \pi[v] \leftarrow u$

În figura 25.3 sunt prezentate două exemple de relaxare a unei muchii, în unul dintre exemple estimarea drumului minim descreşte, în cel de al doilea nici o estimare nu este modificată.

Fiecare algoritm din cadrul acestui capitol apelează INIŢIALIZEAZĂ-SURSĂ-UNICĂ după care trece la relaxarea repetată a muchiilor. Mai mult decât atât, relaxarea este unica modalitate prin care estimările drumurilor minime şi predecesorii sunt modificaţi. Algoritmii din acest capitol diferă în ceea ce priveşte numărul relaxărilor efectuate şi ordinea în care muchiile sunt relaxate. În algoritmul Dijkstra şi în algoritmul pentru determinarea drumurilor minime în grafurile aciclice orientate, fiecare muchie este relaxată o singură dată. În cazul algoritmului Bellman-Ford, fiecare muchie este relaxată de mai multe ori.

Proprietăţi ale operaţiei de relaxare

Corectitudinea algoritmilor din acest capitol depinde de o serie de proprietăţi ale relaxării care sunt prezentate în cadrul următoarelor leme. Majoritatea lemelor se referă la rezultatele efectuării unei secvenţe de paşi de relaxare asupra muchiilor unui graf orientat cu costuri, iniţializat prin INIŢIALIZEAZĂ-SURSĂ-UNICĂ. Cu excepţia lemei 25.9, aceste leme sunt aplicabile oricărei secvenţe de paşi de relaxare, nu numai acelora care produc valorile drumurilor minime.

[1]Poate părea ciudat faptul că termenul de "relaxare" este utilizat pentru a desemna o operaţie care determină decrementarea unei margini superioare. Utilizarea termenului este justificată de considerente de ordin istoric. Rezultatul unui pas de relaxare poate fi văzut ca o relaxare a restricţiei $d[v] \leq d[u] + w(u, v)$ care, prin lema 25.3, trebuie satisfacută dacă $d[u] = \delta(s, u)$ şi $d[v] = \delta(s, v)$. Cu alte cuvinte, dacă $d[v] \leq d[u] + w(u, v)$ atunci nu există "presiune" pentru satisfacerea acestei restricţii, deci restricţia poate fi "relaxată".

Lema 25.4 Fie $G = (V, E)$ un graf orientat cu costuri cu funcţia de cost $w : E \to \mathbb{R}$ şi fie $(u, v) \in E$. Atunci, după relaxarea muchiei (u, v) prin executarea procedurii RELAXEAZĂ(u, v, w), rezultă $d[v] \leq d[u] + w(u, v)$.

Demonstraţie. Dacă înaintea relaxării muchiei (u, v) avem $d[v] > d[u] + w(u, v)$, atunci prin relaxare rezultă $d[v] = d[u] + w(u, v)$. Dacă înaintea aplicării relaxării $d[v] \leq d[u] + w(u, v)$, atunci nici $d[v]$ şi nici $d[u]$ nu se modifică, deci, în continuare, după aplicarea relaxării avem $d[v] \leq d[u] + w(u, v)$. ∎

Lema 25.5 Fie $G = (V, E)$ un graf orientat cu costuri cu funcţia de cost $w : E \to \mathbb{R}$. Fie $s \in V$ un vârf sursă şi presupunem că graful este iniţializat prin INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s). Atunci, $d[v] \geq \delta(s, v)$ pentru orice $v \in V$ şi relaţia se păstrează pentru orice secvenţă de paşi de relaxare efectuaţi asupra muchiilor lui G. Mai mult decât atât, o dată ce $d[v]$ îşi atinge marginea inferioară $\delta(s, v)$, valoarea lui nu se mai modifică.

Demonstraţie. Relaţia $d[v] \geq \delta(s, v)$ este cu certitudine îndeplinită după iniţializare deoarece $d[s] = 0 \geq \delta(s, s)$ (a se observa că $\delta(s, s)$ este $-\infty$ dacă s se află pe un ciclu de cost negativ şi este egală cu 0 altfel) şi $d[v] = \infty$ implică $d[v] \geq \delta(s, v)$ pentru orice $v \in V - \{s\}$. Vom demonstra prin reducere la absurd că relaţia se menţine pentru orice secvenţă de paşi de relaxare. Fie v primul vârf pentru care un pas de relaxare a muchiei (u, v) determină $d[v] < \delta(s, v)$. Atunci, după relaxarea muchiei (u, v) obţinem

$$d[u] + w(u, v) = d[v] < \delta(s, v) \leq \delta(s, u) + w(u, v) \text{ (prin lema 25.3)},$$

ceea ce implică $d[u] < \delta(s, u)$. Deoarece, însă, relaxarea muchiei (u, v) nu modifică $d[u]$, această inegalitate ar fi trebuit să fie adevărată înainte de relaxarea muchiei, ceea ce contrazice alegerea lui v ca fiind primul vârf pentru care $d[v] < \delta(s, v)$. Rezultă astfel că relaţia $d[v] \geq \delta(s, v)$ se păstrează pentru orice $v \in V$.

Pentru a vedea că valoarea lui $d[v]$ nu se mai modifică o dată ce $d[v] = \delta(s, v)$, să observăm că dacă şi-a atins marginea inferioară, deoarece am demonstrat că $d[v] \geq \delta(s, v)$, rezultă că $d[v]$ nu mai poate descreşte şi, de asemenea, $d[v]$ nu poate nici să crească, fiindcă paşii de relaxare nu determină incrementarea valorilor lui d. ∎

Corolarul 25.6 Presupunem că, în graful orientat cu costuri $G = (V, E)$, având funcţia de cost $w : E \to \mathbb{R}$, nu există drum de la vârful sursă $s \in V$ la un vârf dat $v \in V$. Atunci, după iniţializarea grafului prin INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s), avem $d[v] = \delta(s, v)$ şi egalitatea se păstrează pentru orice secvenţă de paşi de relaxare asupra muchiilor lui G.

Demonstraţie. Prin lema 25.5, avem $\infty = \delta(s, v) \leq d[v]$, atunci $d[v] = \infty = \delta(s, v)$. ∎

Următoarea lemă joacă un rol important în demonstrarea corectitudinii algoritmilor pentru determinarea drumurilor minime care vor fi prezentaţi în continuare în acest capitol. Ea va furniza condiţii suficiente pentru ca estimarea unui drum minim, obţinut prin relaxare, să conveargă la costul unui drum minim.

Lema 25.7 Fie $G = (V, E)$ un graf orientat cu costuri cu funcţia de cost $w : E \to \mathbb{R}$. Fie $s \in V$ un vârf sursă şi $s \rightsquigarrow u \to v$ un drum minim în G pentru $u, v \in V$. Presupunem că G este iniţializat cu INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s) după care este efectuată o secvenţă de paşi de relaxare care include apelul procedurii RELAXEAZĂ(u, v, w) asupra muchiilor lui G. Dacă $d[u] = \delta(s, u)$ la un moment care precede acest apel, atunci $d[v] = \delta(s, v)$ la orice moment ulterior acestui apel.

Demonstraţie. Prin lema 25.5, dacă $d[u] = \delta(s,u)$ la un moment dat înainte ca muchia (u,v) să fie relaxată, atunci egalitatea se păstrează în continuare. În particular, după relaxarea muchiei (u,v) avem,

$$\begin{aligned} d[v] &\leq d[u] + w(u,v) \text{ (prin lema 25.4)} \\ &= \delta(s,u) + w(u,v) \\ &= \delta(s,v) \text{ (prin corolarul 25.2).} \end{aligned}$$

Prin lema 25.5, $\delta(s,v)$ este un minorant pentru $d[v]$, deci $d[v] = \delta(s,v)$ şi egalitatea se va menţine la toate momentele ulterioare. ∎

Arbori de drumuri minime

Până în acest moment, am demonstrat că relaxarea determină că estimările drumurilor minime descresc monoton la costurile adevărate corespunzătoare drumurilor minime. De asemenea, vom arăta că, dacă o secvenţă de relaxări a determinat costurile drumurilor minime, subgraful predecesorilor G_π indus de valorile π rezultate este un arbore al drumurilor minime pentru G. Pentru început, următoarea lemă stabileşte că subgraful predecesorilor este un arbore având drept rădăcină vârful sursă considerat.

Lema 25.8 Fie $G = (V,E)$ un graf orientat cu costuri cu funcţia de cost $w : E \to \mathbb{R}$ şi $s \in V$ vârful sursă. Presupunem că G nu are cicluri de cost negativ care să poată fi atinse din s. În aceste condiţii, după ce graful a fost iniţializat prin INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G,s), graful predecesorilor G_π este un arbore cu rădăcină având rădăcina s pentru orice secvenţă de paşi de relaxare aplicată muchiilor lui G.

Demonstraţie. La momentul iniţial singurul vârf din G_π este vârful sursă, deci afirmaţia formulată este evident adevărată. Fie G_π graful predecesorilor rezultat după efectuarea unei secvenţe de paşi de relaxare. Demonstrăm prin reducere la absurd că G_π este graf aciclic. Să presupunem că un anumit pas de relaxare determină apariţia unui ciclu în graful G_π, fie acesta $c = \langle v_0, v_1, ..., v_k \rangle$, $v_k = v_0$. Atunci, $\pi[v_i] = v_{i-1}$ pentru $i = 1,2,...,k$, şi fără restrângerea generalităţii putem presupune că acest ciclu a rezultat în G_π prin relaxarea muchiei (v_{k-1}, v_k).

Afirmăm că toate vârfurile ciclului c pot fi atinse din vârful sursă s. De ce? Fiecare vârf al ciclului c, în momentul în care i-a fost atribuită o valoare diferită de NIL a valorii π, are de asemenea asociată şi o estimare a unui drum minim. Prin lema 25.5, fiecare vârf al ciclului c are costul unui drum minim finit, din care rezultă că vârful este accesibil din s.

Să examinăm acum estimările drumurilor minime ale vârfurilor din c existente înaintea efectuării apelului RELAXEAZĂ(v_{k-1}, v_k, w) şi să arătăm că c este un ciclu de cost negativ, fapt care va contrazice presupunerea că G nu conţine cicluri de cost negativ care să poată fi atinse din vârful sursă. Înainte de efectuarea apelului, avem $\pi[v_i] = v_{i-1}$ pentru $i = 1,2,...,k-1$. Rezultă că pentru $i = 1,2,...,k-1$ ultima reactualizare a valorii $d[v_i]$ a fost prin atribuirea $d[v_i] \leftarrow d[v_{i-1}] + w(v_i, v_{i-1})$. Dacă din acest moment $d[v_{i-1}]$ a fost modificat, atunci noua valoare va fi mai mică, deci imediat înaintea efectuării apelului RELAXEAZĂ(v_{k-1}, v_k, w) avem

$$d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i) \text{ pentru orice } i = 1,2,\ldots,k-1. \tag{25.1}$$

Deoarece $\pi[v_k]$ a fost modificat prin apel, înainte de apel avem inegalitatea strictă

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k).$$

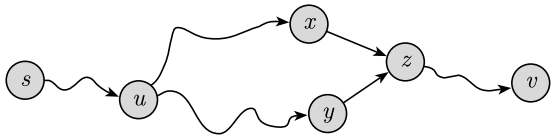


Figura 25.4 Ilustrarea unicităţii drumului în G_π de la sursa s la vârful v. Dacă ar exista două drumuri $p_1(s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v)$ şi $p_2(s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v)$, unde $x \neq y$, atunci $\pi[z] = x$ şi $\pi[z] = y$ ceea ce conduce la o contradicţie.

Prin însumarea acestei inegalităţi stricte cu cele $k-1$ inegalităţi (25.1), obţinem suma estimărilor drumurilor minime de-a lungul ciclului c:

$$\sum_{i=1}^{k} d[v_i] > \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Dar,

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}],$$

deoarece fiecare vârf din ciclul c apare exact o dată în fiecare dintre cele două sume. Aceasta implică

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Astfel suma costurilor de-a lungul ciclului c este negativă, ceea ce reprezintă o contradicţie.

Rezultă astfel că G_π este un graf orientat, aciclic. Pentru a arăta că el este un arbore cu rădăcină având rădăcina s, este suficient (vezi exerciţiul 5.5-3) să arătăm că, pentru fiecare vârf $v \in V_\pi$, există un drum unic de la s la v în G_π.

Trebuie să arătăm mai întâi că pentru orice vârf din V_π există un drum de la s către acel vârf. Mulţimea V_π este formată din vârfurile care au valori π diferite de NIL şi din vârful s. Ideea este de a demonstra prin inducţie că există un drum de la s către toate vârfurile din V_π. Detaliile sunt lăsate pe seama cititorului, în exerciţiul 25.1-6.

Pentru completarea demonstraţiei lemei, trebuie arătat că pentru orice vârf $v \in V_\pi$, există cel mult un drum de la s la v în graful G_π. Să presupunem contrariul. Cu alte cuvinte, să presupunem că există două drumuri simple de la s la un anume vârf v : p_1, care poate fi descompus în $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$ şi p_2 care poate fi descompus în $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$ unde $x \neq y$. (Vezi figura 25.4). În acest caz, rezultă $\pi[z] = x$ şi $\pi[z] = y$ ceea ce conduce la contradicţia $x = y$. În consecinţă, există un singur drum în G_π de la s la v, deci G_π este un arbore al drumurilor minime de rădăcină s. ∎

Acum putem demonstra că, dacă după efectuarea unei secvenţe de paşi de relaxare, tuturor vârfurilor le-au fost asociate adevăratele costuri corespunzătoare drumurile minime, atunci subgraful predecesorilor G_π este un arbore al drumurilor minime.

Lema 25.9 Fie $G = (V, E)$ un graf orientat cu costuri cu funcţia de cost $w : E \rightarrow \mathbb{R}$ şi $s \in V$ vârful sursă. Presupunem că G nu are cicluri de cost negativ care să poată fi atinse din s. Să

efectuăm apelul INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s) şi să presupunem în continuare că a fost efectuată o secvenţă de paşi de relaxare asupra muchiilor lui G care determină $d[v] = \delta(s, v)$ pentru orice $v \in V$. Atunci subgraful predecesorilor G_π este un arbore al drumurilor minime de rădăcină s.

Demonstraţie. Trebuie să demonstrăm că cele trei proprietăţi ale arborilor drumurilor minime sunt îndeplinite de către G_π. Pentru demonstrarea primei proprietăţi, trebuie să arătam că V_π este mulţimea vârfurilor care pot fi atinse din s. Prin definiţie, un cost $\delta(s, v)$ al unui drum minim este finit, dacă şi numai dacă v poate fi atins din s şi deci vârfurile care pot fi atinse din s sunt exact vârfurile care au valori d finite. Dar unui vârf $v \in V - \{s\}$ i-a fost atribuită o valoare finită pentru câmpul $d[v]$ dacă şi numai dacă $\pi[v] \neq$ NIL. Rezultă că vârfurile din V_π sunt exact vârfurile care pot fi atinse din s.

Cea de a doua proprietate rezultă direct din lema 25.8. Mai rămâne de demonstrat ultima proprietate a arborilor drumurilor minime şi anume: pentru orice $v \in V_\pi$, unicul drum elementar $s \overset{p}{\rightsquigarrow} v$ în G_π este un drum minim de la s la v în G. Fie $p = \langle v_0, v_1, ..., v_k \rangle$, unde $v_0 = s$ şi $v_k = v$. Pentru $i = 1, 2, ..., k$, avem simultan relaţiile $d[v_i] = \delta(s, v_i)$ şi $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$ din care rezultă $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Prin însumarea costurilor de-a lungul drumului p obţinem

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) = \delta(s, v_k) - \delta(s, v_0) = \delta(s, v_k).$$

Obţinem astfel $w(p) \leq \delta(s, v_k)$. Deoarece $\delta(s, v_k)$ este o margine inferioară pentru costul oricărui drum de la s la v_k, rezultă că $w(p) = \delta(s, v_k)$ şi deci p este un drum minim de la s la $v = v_k$. ■

Exerciţii

25.1-1 Obţineţi doi arbori ai drumurilor minime pentru graful orientat din figura 25.2 distincţi de arborii deja ilustraţi.

25.1-2 Daţi un exemplu de graf orientat cu costuri $G = (V, E)$ având funcţia de cost $w : E \to \mathbb{R}$ şi sursa s astfel încât G să satisfacă următoarea proprietate: pentru orice muchie $(u, v) \in E$, există un arbore al drumurilor minime de rădăcină s care conţine (u, v) şi există un arbore al drumurilor minime de rădăcină s care nu conţine (u, v).

25.1-3 Extindeţi demonstraţia lemei 25.3 astfel încât să includă şi cazurile în care costurile drumurilor minime sunt ∞ sau $-\infty$.

25.1-4 Fie $G = (V, E)$ un graf orientat cu costuri, cu vârful sursă s şi presupunem că G este iniţializat prin INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s). Demonstraţi că dacă o secvenţă de paşi de relaxare determină ca $\pi[s]$ să fie diferit de NIL, atunci G conţine un ciclu de cost negativ.

25.1-5 Fie $G = (V, E)$ un graf orientat cu costuri, care nu conţine muchii de cost negativ. Fie $s \in V$ vârful sursă şi definim $\pi[v]$ în mod obişnuit şi anume $\pi[v]$ este predecesor al lui v pe un drum minim de la sursa s la $v \in V - \{s\}$ dacă v poate fi atins din s şi NIL în caz contrar. Daţi un exemplu de astfel de graf G şi o atribuire a valorilor π care să producă un ciclu în G_π. (Prin lema 25.8, o astfel de atribuire nu poate fi realizată pe baza unei secvenţe de paşi de relaxare).

25.1-6 Fie $G = (V, E)$ un graf orientat cu costuri cu funcţia de cost $w : E \to \mathbb{R}$ şi care nu are cicluri de cost negativ. Fie $s \in V$ vârful sursă şi presupunem că G este iniţializat prin INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s). Demonstraţi că pentru orice vârf $v \in V_\pi$, există drum de la s la v în G_π şi că această proprietate se păstrează pentru orice secvenţă de paşi de relaxare.

25.1-7 Fie $G = (V, E)$ un graf orientat cu costuri, care nu conţine nici un ciclu de cost negativ, accesibil din vârful sursă s. Fie s vârful sursă, şi să presupunem că G a fost iniţializat de procedura INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s). Arătaţi că există un şir de $|V| - 1$ paşi de relaxare care produce $d[v] = \delta(s, v)$ pentru orice $v \in V$.

25.1-8 Fie G un graf orientat cu costuri oarecare şi cu un ciclu de cost negativ accesibil din vârful sursă s. Arătaţi că întotdeauna se poate construi un şir infinit de relaxări ale muchiilor lui G astfel încât fiecare relaxare face ca estimarea drumului minim să se schimbe.

25.2. Algoritmul Dijkstra

Algoritmul Dijkstra rezolvă problema drumurilor minime de sursă unică într-un graf orientat cu costuri $G = (V, E)$ pentru care toate costurile muchiilor sunt nenegative. În cadrul acestei secţiuni vom presupune că $w(u, v) \geq 0$ pentru fiecare muchie $(u, v) \in E$.

DIJKSTRA(G, w, s)
1: INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s)
2: $S \leftarrow \emptyset$
3: $Q \leftarrow V[G]$
4: **cât timp** $Q \neq \emptyset$ **execută**
5: $u \leftarrow$EXTRAGE-MIN(Q)
6: $S \leftarrow S \bigcup \{u\}$
7: **pentru** fiecare vârf $v \in Adj[u]$ **execută**
8: RELAXEAZĂ(u, v, w)

Algoritmul Dijkstra operează prin menţinerea unei mulţimi S de vârfuri pentru care costurile finale corespunzătoare drumurilor minime de la sursa s au fost deja determinate, respectiv pentru toate vârfurile $v \in S$ avem $d[v] = \delta(s, v)$. Algoritmul iterează selectarea unui vârf $u \in V - S$ pentru care estimarea drumului minim este minimă, introduce u în S şi relaxează muchiile divergente din u. În implementarea care va fi prezentată se va menţine o coadă de priorităţi Q conţinând toate vârfurile din $V - S$ indexate prin valorile d corespunzătoare. Implementarea presupune că graful G este reprezentat prin liste de adiacenţă.

Algoritmul Dijkstra relaxează muchiile în modul prezentat în figura 25.5. Linia 1 realizează iniţializarea obişnuită a valorilor d şi π, iar linia 2 iniţializează mulţimea S cu mulţimea vidă. În continuare, coada de priorităţi Q este iniţializată în linia 3 pentru a conţine toate vârfurile din $V - S = V - \emptyset = V$. La fiecare iteraţie a buclei **cât timp** din liniile 4–8, este extras câte un vârf u din $Q = V - S$ şi introdus în mulţimea S. (La prima iteraţie a acestei bucle $u = s$). Astfel, vârful u are cea mai mică estimare a drumului minim dintre vârfurile din $V - S$. În continuare, liniile 7–8 relaxează fiecare muchie (u, v) divergentă din u şi reactualizează estimarea $d[v]$ şi predecesorul $\pi[v]$ în cazul în care drumul minim către v poate fi ameliorat prin trecerea prin u.

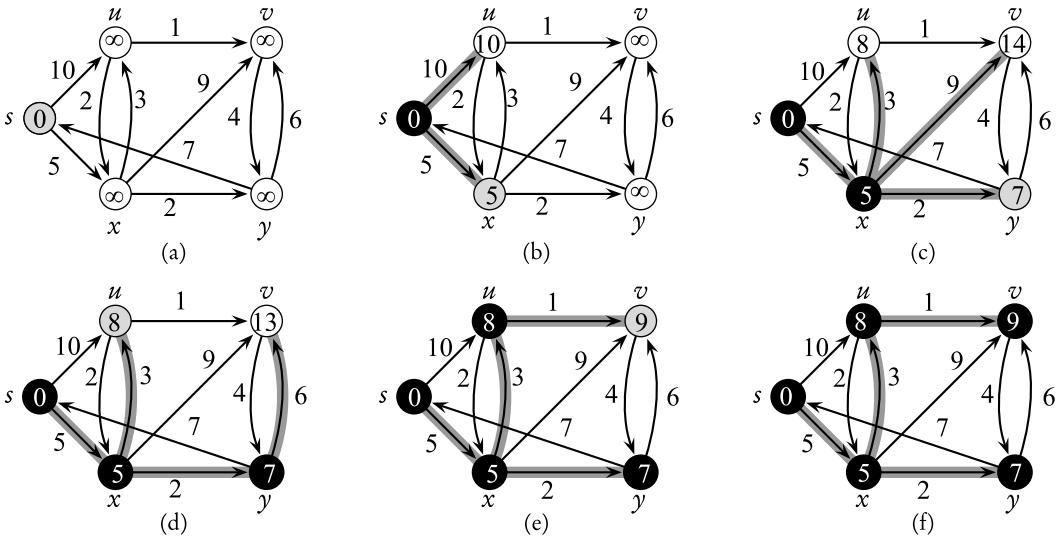


Figura 25.5 Evoluţia algoritmului Dijkstra. Vârful sursă este vârful cel mai din stânga. Estimările drumurilor minime sunt indicate în interiorul vârfurilor, muchiile haşurate reprezintă valorile predecesorilor: dacă muchia (u, v) este haşurată, atunci $\pi[v] = u$. Vârfurile marcate cu neagru aparţin mulţimii S iar vârfurile marcate cu alb aparţin cozii de priorităţi $Q = V - S$. **(a)** Configuraţia existentă imediat înaintea primei iteraţii a buclei **cât timp** din liniile 4–8. Vârful haşurat are valoarea d minimă şi este vârful selectat ca vârf u în linia 5. **(b)-(f)** Configuraţia rezultată după fiecare iteraţie a buclei **cât timp**. Vârful haşurat este selectat ca vârf u la linia 5 a iteraţiei următoare. Valorile d şi π prezentate în (f) sunt valorile finale.

Trebuie observat că nici un vârf nu mai este introdus în Q după linia 3 şi că fiecare vârf este extras din Q şi inserat în S o singură dată, deci bucla **cât timp** din liniile 4–8 efectuează $|V|$ iteraţii.

Deoarece algoritmul Dijkstra alege întotdeauna "cel mai uşor" sau "apropiat" vârf din $V - S$ pentru a-l introduce în S, spunem că el utilizează o strategie de tip greedy. Strategiile greedy sunt prezentate detaliat în capitolul 17, dar considerăm că este posibilă înţelegerea algoritmului Dijkstra chiar dacă acest capitol nu a fost lecturat. În general strategiile greedy nu asigură obţinerea soluţiei optimale, dar aşa după cum va fi demonstrat de teorema următoare şi corolarul acesteia, algoritmul Dijkstra determină într-adevăr drumurile minime. Cheia demonstraţiei este de a arăta că în momentul la care vârful u este introdus în mulţimea S avem $d[u] = \delta(s, u)$ pentru toate vârfurile $u \in V$.

Teorema 25.10 (Corectitudinea algoritmului Dijkstra) Dacă algoritmul Dijkstra este aplicat unui graf orientat, cu costuri $G = (V, E)$ cu funcţie de cost nenegativă w şi vârf sursă s, atunci la terminare $d[u] = \delta(s, u)$ pentru toate vârfurile $u \in V$.

Demonstraţie. Vom demonstra prin reducere la absurd că pentru fiecare vârf $u \in V$, în momentul la care u este introdus în mulţimea S, $d[u] = \delta(s, u)$ şi egalitatea se menţine la toate momentele de timp ulterioare.

Presupunem că u este primul vârf pentru care $d[u] \neq \delta(s, u)$ în momentul la care el este

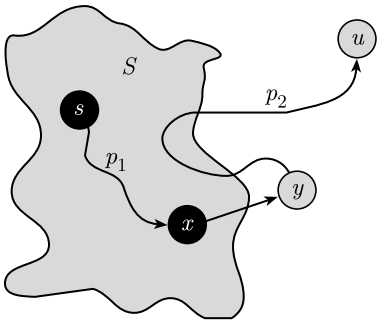


Figura 25.6 Demonstraţia teoremei 25.10. Înainte ca vârful u să fie introdus în S, mulţimea S este nevidă. Un drum minim p de la vârful sursă s la vârful u poate fi descompus în $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$, unde y este primul vârf al drumului care nu aparţine mulţimii S şi $x \in S$ precede y. Vârfurile x şi y sunt distincte, dar este posibil să avem $s = x$ sau $y = u$. Este posibil ca drumul p_2 să reintre în mulţimea S după cum este posibil ca el să nu mai reintre în această mulţime.

introdus în mulţimea S. Să examinăm configuraţia existentă la începutul iteraţiei din bucla **cât timp** care determină introducerea vârfului u în S şi prin examinarea unui drum minim de la s la u să obţinem concluzia $d[u] = \delta(s, u)$. Deoarece s este primul vârf care este introdus în S, şi $d[s] = \delta(s, s) = 0$, în acest moment, rezultă $u \neq s$. Deoarece $u \neq s$, avem $S \neq \emptyset$ în momentul la care u este introdus în S. În mod necesar există un drum de la s la u, deoarece în caz contrar, prin corolarul 25.6 am avea $d[u] = \delta(s, u) = \infty$ ceea ce ar contrazice ipoteza $d[u] \neq \delta(s, u)$. Deoarece există cel puţin un drum de la s la u, există şi un drum minim p de la s la u. Drumul p conectează un vârf s din S cu un vârf u din $V - S$. Fie y primul vârf de-a lungul lui p astfel încât $y \in V - S$ şi fie $x \in V$ predecesorul lui y. Atunci, aşa cum rezultă din figura 25.6, drumul p poate fi descompus ca $s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$.

Afirmăm că $d[y] = \delta(s, y)$ în momentul la care u este introdus în S. Pentru a demonstra această afirmaţie, observăm că $x \in S$. Deoarece u este primul vârf pentru care $d[u] \neq \delta(s, u)$ în momentul la care el este introdus în S, rezultă $d[x] = \delta(s, x)$ în momentul la care x este introdus în S. Muchia (x, y) a fost însă relaxată în acest moment, deci afirmaţia rezultă pe baza lemei 25.7.

Acum putem deriva o contradicţie care să conducă la justificarea concluziei formulate în teoremă. Deoarece y precede u pe un drum minim de la s la u şi toate costurile muchiilor sunt nenegative (în particular şi cele care compun drumul p_2), avem $\delta(s, y) \leq \delta(s, u)$ şi deci

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u] \text{ (prin lema 25.5).} \tag{25.2}$$

Dar, deoarece ambele vârfuri u şi y se aflau deja în $V - S$ în momentul la care u a fost selectat în linia 5, avem $d[u] \leq d[y]$. Astfel, ambele inegalităţi din (25.2) sunt de fapt egalităţi, deci

$$d[y] = \delta(s, y) = \delta(s, u) = d[u].$$

În consecinţă, $d[u] = \delta(s, u)$ ceea ce contrazice alegerea vârfului u. În concluzie, în momentul în care fiecare vârf $u \in V$ este inserat în mulţimea S, avem $d[u] = \delta(s, u)$ şi conform lemei 25.5 egalitatea se menţine la toate momentele ulterioare de timp. ■

Corolarul 25.11 Dacă algoritmul Dijkstra este aplicat unui graf orientat cu costuri $G = (V, E)$ cu funcţie de cost nenegativă w şi vârf sursă s, atunci la terminare subgraful predecesorilor G_π este un arbore al drumurilor minime de rădăcină s.

Demonstraţie. Rezultă direct din teorema 25.10 şi lema 25.9. ∎

Analiză

Cât de rapid este algoritmul Dijkstra? Să considerăm mai întâi cazul în care reprezentăm coada de priorităţi $Q = V - S$ printr-un tablou de tip listă. În cazul unei astfel de implementări, fiecare operaţie EXTRAGE-MIN necesită un timp $O(V)$ şi cum sunt $|V|$ operaţii, timpul total necesar pentru EXTRAGE-MIN este $O(V^2)$. Fiecare vârf $v \in V$ este inserat în mulţimea S o singură dată, deci pe durata evoluţiei algoritmului, fiecare muchie din lista de adiacenţă $Adj[v]$ este examinată de asemenea o singură dată în cadrul buclei **pentru** din liniile 7–8. Deoarece numărul total de muchii în listele de adiacenţă este $|E|$, vor fi în total $|E|$ iteraţii pentru această buclă **pentru**, fiecare operaţie necesitând timp $O(1)$. Timpul total de execuţie al algoritmului este astfel $O(V^2 + E) = O(V^2)$.

Dacă graful este "rar", atunci este preferabilă implementarea listei de priorităţi Q ca heap binar. Algoritmul rezultat este uneori referit ca ***algoritmul Dijkstra modificat***. În acest caz, fiecare operaţie EXTRAGE-MIN necesită timp $O(\lg V)$. Ca şi în varianta iniţială sunt $|V|$ astfel de operaţii. Timpul necesar generării heap-ului binar este $O(V)$. Atribuirile $d[v] \to d[u] + w(u, v)$ în RELAXEAZĂ sunt realizate prin apelul DECREŞTE-CHEIE$(Q, v, d[u] + w(u, v))$ care necesită timp $O(\lg V)$ (vezi exerciţiul 7.5-4) şi sunt cel mult $|E|$ astfel de operaţii. Astfel, timpul de execuţie este $O((V + E) \lg V)$, care devine $O(E \lg V)$ în cazul în care toate vârfurile pot fi atinse din vârful sursă. De fapt, putem obţine un timp de execuţie $O(V \lg V + E)$ implementând coada de priorităţi Q prin intermediul unui heap Fibonacci (vezi capitolul 21). Costul amortizat al fiecăreia dintre cele $|V|$ operaţii EXTRAGE-MIN este $O(\lg V)$ şi fiecare dintre cele $|E|$ apeluri DESCREŞTE-CHEIE necesită numai $O(1)$ timp amortizat. Din punct de vedere istoric, dezvoltarea heap-urilor Fibonacci a fost motivată de observaţia că în algoritmul Dijkstra modificat sunt în general mai multe apeluri DESCREŞTE-CHEIE decât apeluri EXTRAGE-MIN, deci orice metodă de reducere a timpului amortizat a fiecărei operaţii DESCREŞTE-CHEIE la $o(\lg V)$ fără să crească timpul amortizat corespunzator unui apel EXTRAGE-MIN ar determina o implementare asimptotic mai rapidă.

Algoritmul Dijkstra prezintă similitudini atât cu căutarea în lăţime (vezi secţiunea 23.2) cât şi cu algoritmul lui Prim pentru determinarea arborelui parţial minim (vezi secţiunea 24.2). El este similar cu algoritmul de căutare în lăţime, deoarece mulţimea de vârfuri S corespunde mulţimii de vârfuri "negre" utilizate în căutarea în lăţime; exact aşa cum vârfurile din S au costurile finale corespunzătoare drumurilor minime, vârfurilor "negre" din cazul căutării în lăţime le corespund distanţele corecte ale căutării în lăţime. Algoritmul Dijkstra este similar cu algoritmul Prim deoarece ambii algoritmi utilizează cozi de priorităţi pentru găsirea celui mai "uşor" vârf din complementara unei mulţimi date (mulţimea S în algoritmul Dijkstra şi arborele generat de algoritmul Prim), inserează acest vârf într-o anumită mulţime şi ajustează în mod corespunzător costurile vârfurilor rezidente în continuare în complementara mulţimii.

Exerciţii

25.2-1 Aplicaţi algoritmul Dijkstra grafului orientat prezentat în figura 25.2, mai întâi cu s ca vârf sursă şi apoi considerând vârf sursă pe y. În stilul prezentat în figura 25.5, indicaţi valorile d şi π şi vârfurile din mulţimea S după fiecare iteraţie a buclei **cât timp**.

25.2-2 Daţi un exemplu de graf orientat având muchii de costuri negative pentru care algoritmul Dijkstra produce rezultate incorecte. De ce nu mai funcţionează demonstraţia teoremei 25.10 în cazul în care sunt acceptate muchii de costuri negative?

25.2-3 Să presupunem că modificăm linia 4 din algoritmul Dijkstra în modul următor:

4: **cât timp** $|Q| > 1$

Această modificare determină ca bucla **cât timp** să aibă $|V| - 1$ iteraţii în loc de $|V|$. Este acest algoritm corect?

25.2-4 Presupunem că $G = (V, E)$ este un graf orientat astfel încât fiecărei muchii $(u, v) \in E$ îi este asociată o valoare $r(u, v)$, număr real în domeniul $0 \leq r(u, v) \leq 1$ reprezentând fiabilitatea unui canal de comunicaţie de la vârful u la vârful v. Considerăm valoarea $r(u, v)$ ca fiind probabilitatea de a nu se defecta canalul de la vârful u la vârful v. Mai presupunem că aceste probabilităţi sunt independente. Propuneţi un algoritm eficient pentru determinarea celui mai sigur drum între două vârfuri.

25.2-5 Fie $G = (V, E)$ graf orientat cu costuri cu funcţia de cost $w : E \to \{0, 1, ..., W - 1\}$ pentru W număr întreg nenegativ dat. Modificaţi algoritmul Dijkstra astfel încât să determine drumurile minime de sursă s unică în timp $O(WV + E)$.

25.2-6 Modificaţi algoritmul pe care l-aţi propus pentru rezolvarea exerciţiului 25.2-5 astfel încât timpul de execuţie să fie $O((V + E) \lg W)$. (*Indicaţie:* Câte estimări distincte pentru drumurile minime pot exista în timp pentru punctele din $V - S$?)

25.3. Algoritmul Bellman-Ford

Algoritmul Bellman-Ford rezolvă problema drumurilor minime de sursă unică în cazul mai general în care costurile muchiilor pot fi negative. Fiind dat un graf orientat cu costuri $G = (V, E)$ cu vârful sursă s şi funcţia de cost $w : E \to \mathbb{R}$, algoritmul Bellman-Ford returnează o valoare booleană indicând dacă există sau nu un ciclu de cost negativ accesibil din vârful sursă considerat. În cazul în care un astfel de ciclu există, algoritmul semnalează că nu există soluţie, respectiv dacă nu există un astfel de ciclu, algoritmul produce drumurile minime şi costurile corespunzătoare lor.

Ca şi algoritmul Dijkstra, algoritmul Bellman-Ford utilizează tehnica de relaxare, procedând la descreşterea estimării $d[v]$ a drumului minim de la sursă s la fiecare vârf $v \in V$ până când este obţinut costul adevărat $\delta(s, v)$ corespunzător unui drum minim. Algoritmul returnează ADEVĂRAT dacă şi numai dacă nu conţine cicluri de cost negativ accesibile din sursă.

În figura 25.7 este prezentat modul în care operează algoritmul Bellman-Ford pe un graf cu 5 vârfuri. După efectuarea iniţializării, algoritmul efectuează $|V| - 1$ treceri prin muchiile

grafului. Fiecare trecere este o iteraţie a buclei **pentru** din liniile 2–4 şi efectuează relaxarea fiecărei muchii câte o singură dată. Figurile 25.7(b)-(e) prezintă starea algoritmului după fiecare dintre cele patru treceri prin mulţimea muchiilor. După efectuarea a $|V| - 1$ treceri, liniile 5–8 verifică existenţa unui ciclu de cost negativ şi returnează valoarea booleană corespunzătoare. (Vom vedea puţin mai târziu de ce această verificare este operantă.)

BELLMAN-FORD(G, w, s)
1: INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s)
2: **pentru** $i \to 1, |V[G]| - 1$ **execută**
3: **pentru** fiecare muchie $(u, v) \in E[G]$ **execută**
4: RELAXEAZĂ(u, v, w)
5: **pentru** fiecare muchie $(u, v) \in E[G]$ **execută**
6: **dacă** $d[v] > d[u] + w(u, v)$ **atunci**
7: **returnează** FALS
8: **returnează** ADEVĂRAT

Algoritmul Bellman-Ford este de complexitate temporală $O(VE)$ deoarece iniţializarea din linia 1 necesită timp $\Theta(V)$, fiecare dintre cele $|V| - 1$ treceri prin mulţimea muchiilor iniţiate de liniile 2–4 necesită un timp $O(E)$, iar bucla **pentru** din liniile 5–7 se realizează tot în timpul $O(E)$.

Pentru demonstrarea corectitudinii algoritmului Bellman-Ford începem prin a arăta că dacă nu există cicluri de cost negativ, atunci algoritmul determină costurile corecte corespunzătoare drumurilor minime pentru toate vârfurile care pot fi atinse din sursă. Demonstraţia lemei următoare aduce argumentul de ordin intuitiv pe care este bazat acest algoritm.

Lema 25.12 Fie $G = (V, E)$ un graf orientat cu costuri cu vârful sursă s şi funcţia de cost $w : E \to \mathbb{R}$ şi presupunem că G nu conţine cicluri de cost negativ care să poată fi atinşi din s. În aceste condiţii, la terminarea calculului iniţiat de BELLMAN-FORD avem $d[v] = \delta(s, v)$ pentru toate vârfurile v accesibile din s.

Demonstraţie. Fie v un vârf care poate fi atins din s şi fie $p = \langle v_0, v_1, ..., v_k \rangle$ un drum minim de la s la v, unde $v_0 = s$ şi $v_k = v$. Drumul p este simplu, deci $k \leq |V| - 1$. Demonstrăm prin inducţie că pentru a i-a trecere prin muchiile lui G, $i = 0, 1, ..., k$, avem $d[v_i] = \delta(s, v_i)$ şi această egalitate se păstrează la toate momentele de timp ulterioare. Deoarece numărul total de treceri este $|V| - 1$, stabilirea acestei proprietăţi este suficientă pentru demonstraţie. După iniţializare avem $d[v_0] = \delta(s, v_0) = 0$ şi prin lema 25.5, egalitatea se menţine în continuare. Pentru demonstrarea pasului inductiv, presupunem că după cea de a $(i - 1)$-a trecere, $d[v_{i-1}] = \delta(s, v_{i-1})$. Muchia (v_{i-1}, v_i) este relaxată la cea de a i-a trecere, deci prin lema 25.7 rezultă $d[v_i] = \delta(s, v_i)$ după cea de a $i-a$ trecere şi la toate momentele de timp ulterioare, ceea ce încheie demonstraţia. ■

Corolarul 25.13 Fie $G = (V, E)$ un graf orientat cu costuri, cu vârful sursă s şi funcţia de cost $w : E \to \mathbb{R}$. Atunci, pentru fiecare vârf $v \in V$, există drum de la s la v dacă şi numai dacă prin aplicarea algoritmului BELLMAN-FORD asupra grafului G, la terminare avem $d[v] < \infty$.

Demonstraţie. Demonstraţia este similară cu cea a lemei 25.12 şi este propusă spre rezolvare în exerciţiul 25.3-2. ■

Teorema 25.14 (Corectitudinea algoritmului Bellman-Ford) Presupunem că algoritmul BELLMAN-FORD este aplicat grafului orientat $G = (V, E)$ cu vârful sursă s şi funcţia de cost

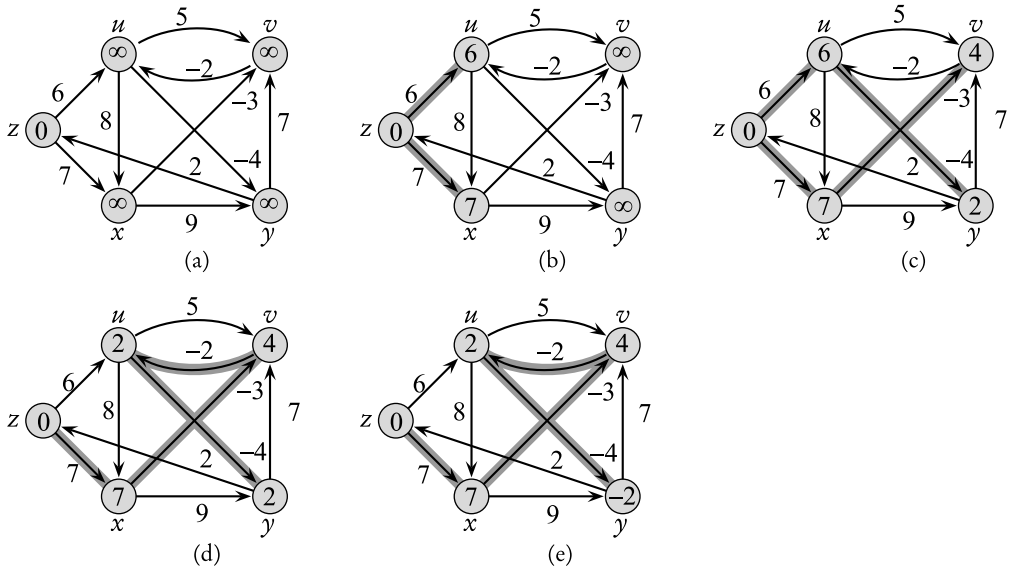


Figura 25.7 Evoluţia determinată de algoritmul Bellman-Ford. Sursa este vârful z. Valorile d sunt indicate în interiorul vârfurilor, iar muchiile haşurate indică valorile π. În acest exemplu, la fiecare trecere muchiile sunt relaxate în ordine lexicografică: (u, v), (u, x), (u, y), (v, u), (x, v), (x, y), (y, v), (y, z), (z, u), (z, x). **(a)** Starea imediat anterioară primei treceri prin mulţimea muchiilor. **(b)-(e)** Starea după fiecare dintre trecerile consecutive prin mulţimea muchiilor. Valorile d şi π din (e) sunt valorile finale. Pentru acest exemplu, algoritmul Bellman-Ford returnează ADEVĂRAT.

$w : E \to \mathbb{R}$. Dacă G nu conţine cicluri de cost negativ, accesibile din s, atunci algoritmul returnează ADEVĂRAT şi avem $d[v] = \delta(s, v)$ pentru toate vârfurile $v \in V$, iar subgraful predecesorilor G_π este un arbore al drumurilor minime cu rădăcină s. Dacă G conţine cel puţin un ciclu de cost negativ, accesibil din s, atunci algoritmul returnează FALS.

Demonstraţie. Presupunem că graful G nu conţine cicluri de cost negativ care pot fi atinşi din sursa s. Demonstrăm mai întâi că la terminare $d[v] = \delta(s, v)$ pentru toate vârfurile $v \in V$. Dacă vârful v este accesibil din s, atunci afirmaţia formulată rezultă pe baza lemei 25.12. Dacă v nu este accesibil din s, atunci afirmaţia rezultă din corolarul 25.6. Rezultă că afirmaţia formulată este demonstrată pentru toate vârfurile grafului. Lema 25.9, împreună cu această proprietate, implică faptul că G_π este un arbore al drumurilor minime. Să folosim acum această proprietate pentru a demonstra că BELLMAN-FORD returnează ADEVĂRAT. La terminare, avem pentru toate muchiile $(u, v) \in E$,

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \text{(prin lema 25.3)} \\ &= d[u] + w(u, v), \end{aligned}$$

şi ca atare niciunul din testele din linia 6 nu va determina ca BELLMAN-FORD să returneze FALS, deci va fi returnat ADEVĂRAT.

Reciproc, să presupunem că graful G conţine un ciclu $c = \langle v_0, v_1, ..., v_k \rangle$ de cost negativ, accesibil din sursa s, unde $v_0 = v_k$. Atunci

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0. \tag{25.3}$$

Să presupunem prin absurd că algoritmul BELLMAN-FORD returnează ADEVĂRAT. Atunci $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ pentru $i = 1, 2, ..., k$. Însumând aceste relaţii de-a lungul ciclului c obţinem

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

Ca şi în cazul demonstraţiei lemei 25.8, fiecare vârf din c apare exact o singură dată în fiecare dintre cele două sume, deci

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}].$$

Deoarece prin corolarul 25.13, $d[v_i]$ este finită pentru $i = 1, 2, ..., k$, rezultă

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

ceea ce contrazice inegalitatea (25.3). În concluzie, algoritmul BELLMAN-FORD returnează ADEVĂRAT dacă graful G nu conţine cicluri de cost negativ, accesibile din sursă s şi FALS în caz contrar. ∎

Exerciţii

25.3-1 Aplicaţi algoritmul Bellman-Ford grafului orientat reprezentat în figura 25.7, utilizând vârful y ca vârf sursă. Să se relaxeze muchiile în ordine lexicografică la fiecare trecere şi să se afişeze valorile d şi π după fiecare trecere. În continuare, modificaţi costul muchiei (y, v) astfel încât noul cost să fie egal cu 4 şi aplicaţi din nou algoritmul, utilizând vârful z ca vârf sursă.

25.3-2 Demonstraţi corolarul 25.13.

25.3-3 Fiind dat un graf $G = (V, E)$ orientat, cu costuri, care nu conţine cicluri de cost negativ, fie m valoarea maximă peste toate perechile de vârfuri $u, v \in V$ a numărului minim de muchii care compun un drum minim de la u la v. (Un drum minim se referă la cost şi nu la numărul de muchii.) Propuneţi o modificare simplă a algoritmului Bellman-Ford pentru calculul valorii m în $m + 1$ treceri.

25.3-4 Modificaţi algoritmul Bellman-Ford astfel încât să atribuie $-\infty$ ca valoare pentru $d[v]$ tuturor vârfurilor v cu proprietatea că există un ciclu de cost negativ pe cel puţin un drum de la sursă la v.

25.3-5 Fie $G = (V, E)$ un graf orientat cu costuri, cu funcţia de cost $w : E \to \mathbb{R}$. Obţineţi un algoritm de timp $O(VE)$ pentru determinarea valorii $\delta^*(v) = \min_{u \in V}\{\delta(u, v)\}$, pentru fiecare vârf $v \in V$.

25.3-6 ⋆ Presupunem că graful orientat cu costuri $G = (V, E)$ are un ciclu de cost negativ. Propuneţi un algoritm eficient pentru listarea vârfurilor unui astfel de ciclu. Demonstraţi corectitudinea algoritmului propus.

25.4. Drumuri minime de sursă unică în grafuri orientate aciclice

Prin relaxarea muchiilor unui graf orientat aciclic cu costuri $G = (V, E)$, putem determina drumurile minime de sursă unică în timp $\Theta(V + E)$. Drumurile minime sunt întotdeauna bine definite într-un astfel de graf datorită faptului că acesta nu poate avea cicluri de cost negativ, chiar dacă graful conţine muchii de cost negativ.

Algoritmul începe prin sortarea topologică a grafului (vezi secţiunea 23.4) pentru impunerea unei ordini liniare a vârfurilor. Dacă există un drum de la vârful u la vârful v atunci u precede v în ordinea topologică. În continuare, se efectuează o singură trecere peste vârfurile sortate topologic şi pe măsură ce fiecare vârf este procesat, sunt relaxate toate muchiile divergente din acel vârf.

GOA-DRUMURI-MINIME(G, w, s)
1: sortează în ordine topologică vârfurile din G
2: INIŢIALIZEAZĂ-SURSĂ-UNICĂ(G, s)
3: **pentru** fiecare vârf u în ordinea dată de sortarea topologică **execută**
4: **pentru** fiecare vârf $v \in Adj[u]$ **execută**
5: RELAXEAZĂ(u, v, w)

Un exemplu de aplicare a acestui algoritm este prezentat în figura 25.8.

Timpul de execuţie al acestui algoritm este determinat de linia 1 şi de bucla **pentru** din liniile 3–5. Conform demonstraţiei din secţiunea 23.4, sortarea topologică poate fi realizată în timp $\Theta(V + E)$. În bucla **pentru** din liniile 3–5, ca şi în algoritmul Dijkstra, este efectuată o singură iteraţie pentru fiecare vârf. Pentru fiecare vârf, fiecare muchie divergentă din acel vârf este examinată o singură dată. Spre deosebire de algoritmul Dijkstra, nu există nici o coadă de priorităţi şi, astfel, pentru fiecare muchie este necesar un timp $O(1)$. În concluzie, timpul total necesar este $\Theta(V + E)$, deci algoritmul este liniar în dimensiunea unei reprezentări prin lista de adiacenţă a grafului.

Următoarea teoremă stabileşte că procedura GOA-DRUMURI-MINIME determină corect drumurile minime.

Teorema 25.15 Dacă graful orientat cu costuri $G = (V, E)$ şi de sursă s nu are cicluri, atunci la terminarea procedurii GOA-DRUMURI-MINIME, $d[v] = \delta(s, v)$ pentru toate vârfurile $v \in V$ şi subgraful predecesorilor G_π este un arbore al drumurilor minime.

Demonstraţie. Începem prin a demonstra că la terminare $d[v] = \delta(s, v)$ pentru toate vârfurile $v \in V$. Dacă v nu este accesibil din s, atunci pe baza corolarului 25.6 rezultă $d[v] = \delta(s, v) = \infty$. Să presupunem că v este accesibil din s deci există un drum minim $p = \langle v_0, v_1, ..., v_k \rangle$ unde $v_0 = s$ şi $v_k = v$. Deoarece vârfurile sunt procesate în ordine topologică, muchiile lui p vor fi procesate în ordinea $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$. Utilizând un argument de tip inductiv şi

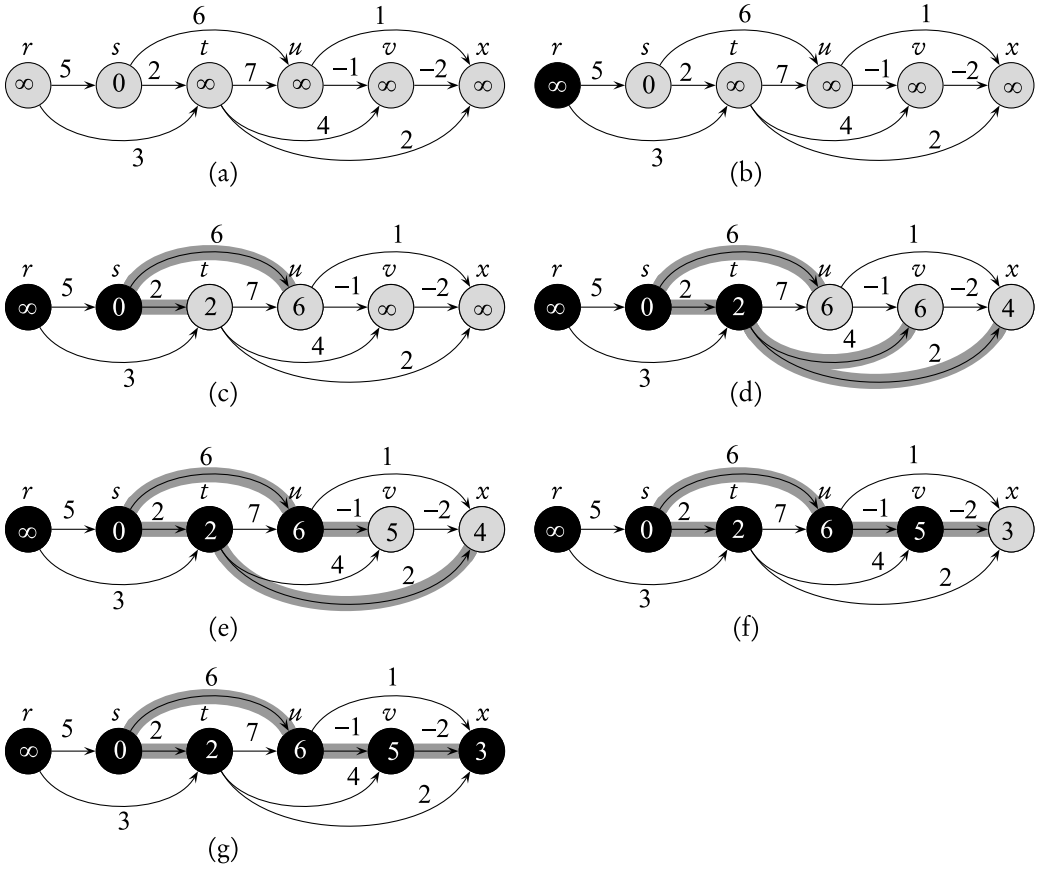


Figura 25.8 Aplicarea algoritmului pentru determinarea drumurilor minime ale unui graf aciclic orientat. Vârfurile sunt sortate topologic de la stânga la dreapta. Vârful sursă este s. Valorile d sunt indicate în interiorul vârfurilor, iar muchiile haşurate reprezintă valorile π. **(a)** Starea existentă înainte de prima iteraţie a buclei **pentru** din liniile 3–5. **(b)-(g)** Stările rezultate după fiecare iteraţie a buclei **pentru** din liniile 3–5. Vârful marcat la fiecare iteraţie a fost utilizat ca vârf u la iteraţia respectivă. Valorile finale sunt prezentate în (g).

utilizând lema 25.7 (similar demonstraţiei lemei 25.12) obţinem la terminare, $d[v_i] = \delta(s, v_i)$ pentru $i = 0, 1, ..., k$. În final, pe baza lemei 25.9, rezultă că G_π este un arbore al drumurilor minime. ■

O aplicaţie interesantă a acestui algoritm este în determinarea drumurilor critice în analiza ***graficelor PERT***[2]. Muchiile reprezintă lucrările ce trebuie efectuate, iar costurile exprimă timpii necesari efectuării diferitelor lucrări. Dacă muchia (u, v) intră în vârful v şi muchia (v, x) iese din vârful v, atunci efectuarea lucrării (u, v) trebuie realizată înaintea lucrării (v, x). Un drum în acest graf orientat aciclic reprezintă o anumită secvenţă de lucrări şi ordinea în care ele trebuie efectuate. Un ***drum critic*** este un *cel mai lung* drum în acest graf orientat aciclic,

[2]"PERT" este un acronim pentru "**p**rogram **e**valuation and **r**eview **t**echnique"

reprezentând timpul maxim necesar efectuării unei secvenţe de lucrări într-o anumită ordine. Costul unui drum critic este o margine inferioară a timpului total necesar efectuării tuturor lucrărilor. Putem determina un drum critic fie

- asociind muchiilor costurile rezultate prin amplificarea cu (-1) a costurilor iniţiale şi aplicând GOA-Drumuri-minime sau
- aplicând GOA-Drumuri-minime în care înlocuim "∞" cu "$-\infty$" în linia 2 a procedurii Iniţializează-Sursă-Unică şi "$>$" prin "$<$" în procedura Relaxează.

Exerciţii

25.4-1 Aplicaţi algoritmul GOA-Drumuri-minime grafului orientat prezentat în figura 25.8 utilizând ca sursă vârful r.

25.4-2 Presupunem că modificăm linia 3 din GOA-Drumuri-minime în modul următor:

3: **pentru** primele $|V| - 1$ vârfuri luate în ordine topologică

Demonstraţi că procedura rezultată este de asemenea corectă.

25.4-3 Analiza PERT prezentată mai sus este destul de neintuitivă. Ar fi mult mai natural ca vârfurile să reprezinte lucrările, iar muchiile restricţiile asupra ordinii în care acestea trebuie să fie efectuate, adică muchia (u, v) indică faptul că lucrarea u trebuie efectuată înaintea lucrării v. În acest caz costurile nu vor mai fi asociate muchiilor, ci vor fi asociate vârfurilor. Modificaţi procedura GOA-Drumuri-Minime astfel încât să determine în timp liniar un cel mai lung drum într-un graf orientat aciclic cu vârfurile având costuri date.

25.4-4 Propuneţi un algoritm eficient pentru calculul numărului total de drumuri într-un graf orientat şi aciclic. Analizaţi algoritmul propus şi comentaţi asupra aspectelor de ordin practic.

25.5. Constrângeri cu diferenţe şi drumurile minime

În cadrul unei probleme de programare liniară, în general, dorim să optimizăm o funcţie liniară supusă unor constrângeri formulate matematic prin intermediul unui sistem de inecuaţii liniare. În cadrul acestei secţiuni vom investiga un caz special de programare liniară care poate fi redus la determinarea drumurilor minime de sursă unică. Problema determinării drumurilor minime de sursă unică, astfel rezultată, poate fi rezolvată utilizând algoritmul Bellman-Ford, obţinând astfel soluţia problemei de programare liniară considerată.

Programare liniară

Într-o problemă generală de ***programare liniară*** sunt date o matrice A de dimensiune $m \times n$, un vector b de dimensiune m şi un vector c de dimensiune n. Se doreşte determinarea unui vector x cu n componente care maximizează ***funcţia obiectiv*** $\sum_{i=1}^{n} c_i x_i$ supusă la m constrângeri date de $Ax \leq b$.

Multitudinea problemelor care pot fi formulate ca programe liniare justifică interesul deosebit acordat obţinerii de algoritmi pentru programarea liniară. În practică, ***algoritmul simplex***[3] rezolvă rapid programe liniare, dar pot fi imaginate date de intrare pentru care algoritmul să necesite timp de execuţie exponenţial. Programele liniare generale pot fi rezolvate în timp polinomial fie prin ***algoritmul elipsoidului***, care însă în practică se dovedeşte a fi destul de lent, fie prin ***algoritmul Karmarkar*** care, în mod frecvent, în practică, este competitiv cu metoda simplex.

Datorită dezvoltărilor de ordin matematic, necesare pentru înţelegerea şi analiza acestor algoritmi, în cadrul secţiunii de faţă nu vor fi prezentaţi algoritmii de programare liniară generali. Totuşi, din mai multe motive este importantă înţelegerea elementelor esenţiale în cadrul abordărilor pe baza tehnicilor de programare liniară. În primul rând, dacă se cunoaşte ca o anumită problemă poate fi identificată ca o problemă de programare liniară de tip polinomial, rezultă că există un algoritm de complexitate temporală polinomială pentru rezolvarea acestei probleme. În al doilea rând, există o serie de cazuri speciale de probleme de programare liniară pentru care sunt cunoscuţi algoritmi mai rapizi. De exemplu, conform consideraţiilor care urmează să fie efectuate în cadrul acestei secţiuni, problema drumurilor minime de sursă unică este un caz special de programare liniară. Din clasa problemelor ce pot fi identificate ca probleme de programare liniară menţionăm problema drumului minim de extremităţi date (exerciţiul 25.5-4) şi problema fluxului maxim (exerciţiul 27.1-8).

În anumite cazuri, de fapt, nu ne interesează funcţia obiectiv şi dorim doar să determinăm o ***soluţie admisibilă*** oarecare, adică orice vector x care verifică sistemul $Ax \leq b$ sau să demonstrăm că nu există soluţii admisibile. În cele ce urmează ne vom concentra pe o astfel de ***problemă de admisibilitate***.

Sisteme de constrângeri cu diferenţe

Într-un ***sistem de constrângeri cu diferenţe***, fiecare linie a matricei A are toate componentele egale cu 0, cu excepţia a două componente, una dintre ele egală cu 1 şi cealaltă egală cu -1. În acest fel, constrângerile reprezentate de sistemul $Ax \leq b$ revin la m ***constrângeri cu diferenţe*** asupra a n necunoscute, fiecare constrângere fiind o inegalitate de expresie

$$x_j - x_i \leq b_k$$

unde $1 \leq i, j \leq n$ şi $1 \leq k \leq m$.

De exemplu, să considerăm problema determinării unui vector 5-dimensional $x = (x_i)$ astfel încât să satisfacă

[3] Algoritmul simplex determină o soluţie optimală a unei probleme de programare liniară prin examinarea unei secvenţe de puncte în regiunea admisibilă – submulţimea spaţiului n-dimensional, soluţie a sistemului $Ax \leq b$. Algoritmul se bazează pe faptul că o soluţie care maximizează funcţia obiectiv în regiunea admisibilă este plasată într-un "punct extrem" sau "colţ" al regiunii admisibile. Algoritmul simplex determină trecerea de la un vârf la altul al regiunii admisibile până când nu se mai realizează nici o ameliorare a funcţiei obiectiv. Un "simplex" este închiderea convexă (vezi secţiunea 35.3) a $d + 1$ puncte în spaţiul d-dimensional (de exemplu, triunghiul în plan sau tetraedrul în spaţiul cu 3 dimensiuni). Conform cu Dantzig [53], operaţia de deplasare de la un vârf la altul poate fi văzută ca o operaţie pe un simplex derivat dintr-o interpretare "duală" a problemei de programare liniară – ceea ce justifică termenul de "metoda simplex".

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

Această problemă este echivalentă cu problema determinării valorilor x_i, $i = 1, 2, ..., 5$ astfel încât următoarele 8 constrângeri cu diferenţe să fie îndeplinite,

$$\begin{array}{rcr} x_1 - x_2 & \leq & 0, \\ x_1 - x_5 & \leq & -1, \\ x_2 - x_5 & \leq & 1, \\ x_3 - x_1 & \leq & 5, \\ x_4 - x_1 & \leq & 4, \\ x_4 - x_3 & \leq & -1, \\ x_5 - x_3 & \leq & -3, \\ x_5 - x_4 & \leq & -3. \end{array} \tag{25.4}$$

Deoarece $x = (-5, -3, 0, -1, -4)$ verifică fiecare dintre inegalităţile precedente, rezultă că x este o soluţie a acestei probleme. De fapt, problema admite mai mult de o singură soluţie; de exemplu $x' = (0, 2, 5, 4, 1)$ este de asemenea soluţie a acestei probleme. Se observă că între aceste două soluţii există o anumită legătură şi anume fiecare componenta a lui x' este suma dintre componenta pereche din x şi numărul 5 şi acest fapt nu este o simplă coincidenţă.

Lema 25.16 Fie $x = (x_1, x_2, ..., x_n)$ o soluţie a sistemului de constrângeri cu diferenţe $Ax \leq b$ şi d o constantă arbitrară. Atunci $x + d = (x_1 + d, x_2 + d, ..., x_n + d)$ este de asemenea o soluţie a sistemului $Ax \leq b$.

Demonstraţie. Pentru fiecare x_i şi x_j avem $(x_j + d) - (x_i + d) = x_j - x_i$. Deci, dacă x satisface sistemul $Ax \leq b$ atunci şi $x + d$ îl satisface. ∎

Sistemele de constrângeri cu diferenţe apar în cadrul multor aplicaţii practice. De exemplu, necunoscutele x_i pot reprezenta momentele de timp la care survin anumite evenimente. Fiecare constrângere poate exprima faptul că un eveniment nu poate apărea mult mai târziu decât un alt eveniment. Posibil ca evenimentele să corespundă lucrărilor ce trebuie executate pe durata construcţiei unei clădiri. Dacă săpatul fundaţiei clădirii începe în momentul x_1 şi necesită 3 zile şi turnatul betonului pentru fundaţie începe în momentul x_2, atunci poate fi formulată constrângerea $x_2 \geq x_1 + 3$ ceea ce poate fi scrisă în mod echivalent $x_1 - x_2 \leq -3$. Cu alte cuvinte, constrângerea relativă în momentele începerii celor două acţiuni poate fi exprimată ca o constrângere cu diferenţe.

Grafuri de constrângere

Este preferabilă interpretarea sistemului de constrângeri cu diferenţe şi din punctul de vedere al teoriei grafurilor. Ideea este că într-un sistem de constrângeri cu diferenţe $Ax \leq b$, matricea A

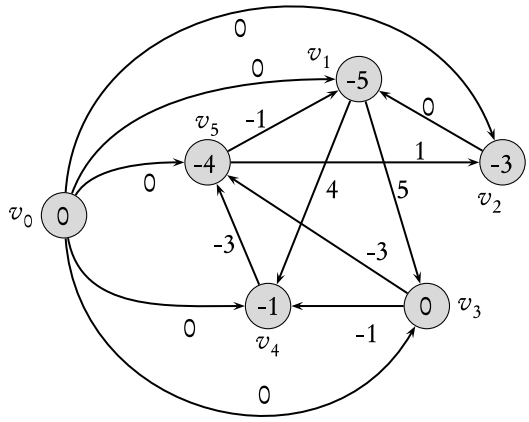


Figura 25.9 Graful constrângerilor corespunzător sistemului de constrângeri cu diferenţe (25.4). Valoarea $\delta(v_0, v_i)$ este reprezentată în fiecare vârf v_i.O soluţie admisibilă a sistemului este $x = (-5, -3, 0, -1, -4)$.

având dimensiunea $n \times m$ poate fi interpretată ca matricea de incidenţă (vezi exerciţiul 23.1-7) a unui graf având n vârfuri şi m muchii. Fiecare vârf v_i al grafului, $i = 1, 2, ..., n$ corespunde uneia dintre cele n variabile necunoscute x_i şi fiecare muchie orientată corespunde uneia dintre cele m inegalităţi în care sunt implicate două necunoscute.

Exprimat în termeni formali, unui sistem de constrângeri cu diferenţe $Ax \leq b$, i se ataşează ***graful constrângerilor***, un graf orientat cu costuri $G = (V, E)$ unde

$$V = \{v_0, v_1, ..., v_n\}$$

şi

$$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ este o constrângere}\} \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), ..., (v_0, v_n)\}.$$

Vârful suplimentar v_0 este introdus, conform discuţiei următoare, pentru a garanta că orice alt vârf este accesibil din v_0. Astfel, mulţimea de vârfuri V constă din câte un vârf v_i pentru fiecare necunoscută x_i şi vârful suplimentar v_0. Mulţimea de muchii E conţine câte o muchie pentru fiecare constrângere cu diferenţe şi în plus câte o muchie (v_0, v_i) pentru fiecare necunoscută x_i. Dacă $x_j - x_i \leq b_k$ este o constrângere cu diferenţe, atunci costul muchiei (v_i, v_j) este $w(v_i, v_j) = b_k$. Costul fiecărei muchii divergente din v_0 este egal cu 0. În figura 25.9 este reprezentat graful constrângerilor pentru sistemul de constrângeri cu diferenţe (25.4).

Următoarea teoremă stabileşte că o soluţie a sistemului de constrângeri cu diferenţe poate fi obţinută prin determinarea costurilor drumurilor minime în graful constrângerilor ataşat.

Teorema 25.17 Fie dat un sistem de constrângeri cu diferenţe $Ax \leq b$ şi fie $G = (V, E)$ graful constrângerilor ataşat. Dacă G nu conţine cicluri de cost negativ, atunci

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), ..., \delta(v_0, v_n)) \tag{25.5}$$

este o soluţie admisibilă a sistemului. Dacă G conţine cicluri de cost negativ, atunci sistemul nu are soluţii admisibile.

Demonstraţie. Demonstrăm mai întâi că, dacă graful constrângerilor nu conţine cicluri de cost negativ, atunci x definit de (25.5) este o soluţie admisibilă. Fie o muchie arbitrară $(v_i, v_j) \in E$. Prin lema 25.3, $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$, sau echivalent, $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$. Rezultă că pentru $x_i = \delta(v_0, v_i)$ şi $x_j = \delta(v_0, v_j)$ constrângerea $x_j - x_i \leq w(v_i, v_j)$ corespunzătoare muchiei (v_i, v_j) este satisfăcută.

Să demonstrăm acum că dacă graful constrângerilor are cicluri de cost negativ, atunci sistemul de constrângeri cu diferenţe nu are soluţie. Fără restrângerea generalităţii putem presupune că ciclul de cost negativ este $c = \langle v_1, v_2, ..., v_k \rangle$, unde $v_1 = v_k$. (Vârful v_0 nu poate aparţine ciclului c deoarece nu există muchii care intră în acest vârf.) Atunci, ciclul c corespunde următoarelor constrângeri cu diferenţe:

$$\begin{aligned} x_2 - x_1 &\leq w(v_1, v_2), \\ x_2 - x_2 &\leq w(v_2, v_3), \\ &\vdots \\ x_k - x_{k-1} &\leq w(v_{k-1}, v_k), \\ x_1 - x_k &\leq w(v_k, v_1). \end{aligned}$$

Deoarece orice soluţie x trebuie să satisfacă fiecare dintre aceste k inegalităţi, orice soluţie va satisface şi inegalitatea rezultată prin însumarea acestora. Deorece în membrul stâng fiecare necunoscută x_i este o dată adunată şi o dată scăzută, prin însumare, membrul stâng este egal cu 0. Prin însumarea membrilor drepţi rezultă $w(c)$ deci obţinem $0 \leq w(c)$. Deoarece c este un ciclu de cost negativ $w(c) < 0$, deci orice soluţie trebuie să satisfacă $0 \leq w(c) < 0$ ceea ce este imposibil. ■

Rezolvarea sistemelor de constrângeri cu diferenţe

Teorema 25.17 permite aplicarea algoritmului Bellman-Ford pentru rezolvarea sistemelor de constrângeri cu diferenţe. Deoarece vârful sursă v_0 este conectat cu fiecare dintre celelalte vârfuri din graful constrângerilor, rezultă că orice ciclu de cost negativ este accesibil din v_0. Astfel, dacă algoritmul Bellman-Ford returnează ADEVĂRAT, atunci costurile unui drum minim determină o soluţie admisibilă a sistemului. De exemplu, în figura 25.9 costurile unui drum minim determină soluţia $x = (-5, -3, 0, -1, -4)$ şi prin lema 25.16, $x = (d - 5, d - 3, d, d - 1, d - 4)$ este de asemenea o soluţie, pentru orice constantă d. Dacă algoritmul Bellman-Ford returnează FALS, atunci sistemul de constrângeri cu diferenţe nu are soluţie admisibilă.

Graful constrângerilor ataşat unui sistem de constrângeri cu diferenţe cu m constrângeri şi n necunoscute are $n + 1$ vârfuri şi $n + m$ muchii. Rezultă că pe baza algoritmului Bellman-Ford, sistemul poate fi rezolvat în timp $O((n + 1)(n + m)) = O(n^2 + nm)$. Exerciţiul 25.5-5 vă cere să demonstraţi că în realitate acest algoritm se execută în timp $O(nm)$ chiar dacă m este mult mai mic decât n.

Exerciţii

25.5-1 Determinaţi o soluţie admisibilă sau decideţi că nu există soluţie pentru următorul sistem de constrângeri cu diferenţe:

$$\begin{aligned}
x_1 - x_2 &\leq 1,\\
x_1 - x_4 &\leq -4,\\
x_2 - x_3 &\leq 2,\\
x_2 - x_5 &\leq 7,\\
x_2 - x_6 &\leq 5,\\
x_3 - x_6 &\leq 10,\\
x_4 - x_2 &\leq 2,\\
x_5 - x_1 &\leq -1,\\
x_5 - x_4 &\leq 3,\\
x_6 - x_3 &\leq -8.
\end{aligned}$$

25.5-2 Determinaţi o soluţie admisibilă sau decideţi că nu există soluţie pentru următorul sistem de constrângeri cu diferenţe:

$$\begin{aligned}
x_1 - x_2 &\leq 4,\\
x_1 - x_5 &\leq 5,\\
x_2 - x_4 &\leq -6,\\
x_3 - x_2 &\leq 1,\\
x_4 - x_1 &\leq 3,\\
x_4 - x_3 &\leq 5,\\
x_4 - x_5 &\leq 10,\\
x_5 - x_3 &\leq -4,\\
x_5 - x_4 &\leq -8.
\end{aligned}$$

25.5-3 Este posibil ca toate drumurile minime de la vârful v_0 într-un graf de constrângeri să fie pozitive? Justificaţi.

25.5-4 Exprimaţi problema drumului minim de extremităţi date ca un program liniar.

25.5-5 Arătaţi cum poate fi modificat algoritmul Bellman-Ford, astfel încât, aplicat pentru rezolvarea unui sistem de constrângeri cu diferenţe de m inegalităţi şi n necunoscute, să se execute în timp $O(mn)$.

25.5-6 Arătaţi cum poate fi rezolvat un sistem de constrângeri cu diferenţe pe baza algoritmului Bellman-Ford aplicat unui graf care nu are vârful suplimentar v_0.

25.5-7 $\star$ Fie $Ax \leq b$ un sistem de m constrângeri cu diferenţe asupra n necunoscute. Arătaţi că algoritmul Bellman-Ford, aplicat grafului constrângerilor maximizează $\sum_{i=1}^{n} x_i$ pe domeniul $Ax \leq b, x_i \leq 0$, pentru orice x_i.

25.5-8 $\star$ Arătaţi că algoritmul Bellman-Ford aplicat grafului constrângerilor ataşat sistemului de constrângeri cu diferenţe $Ax \leq b$, minimizează $(\max\{x_i\} - \min\{x_i\})$ pe domeniul $Ax \leq b$. Explicaţi utilitatea acestei proprietăţi în cazul în care algoritmul este aplicat pentru planificarea lucrărilor în construcţii.

25.5-9 Presupunem că fiecare linie a matricei A a programului liniar $Ax \leq b$ corespunde unei constrângeri cu diferenţe, de o singură variabilă, în forma $x_i \leq b_k$ sau în forma $-x_i \leq b_k$. Arătaţi că algoritmul Bellman-Ford poate fi adaptat pentru rezolvarea acestui tip de sistem de constrângeri.

25.5-10 Să presupunem că vrem să adăugăm şi constrângeri de tip egalitate $x_i = x_j + b_k$ la un sistem de constrângeri cu diferenţe. Arătaţi cum poate fi adaptat algoritmul Bellman-Ford pentru rezolvarea acestui tip de constrângeri.

25.5-11 Propuneţi un algoritm eficient pentru rezolvarea unui sistem de constrângeri cu diferenţe $Ax \leq b$ unde toate componentele vectorului b sunt numere reale şi toate valorile x_i trebuie să fie numere întregi.

25.5-12 $\star$ Propuneţi un algoritm eficient pentru rezolvarea unui sistem de constrângeri cu diferenţe $Ax \leq b$ unde toate componentele vectorului b sunt numere reale şi o parte, dar nu în mod necesar toate, dintre valorile x_i trebuie să fie numere întregi.

Probleme

25-1 Varianta Yen a algoritmului Bellman-Ford
Să presupunem că, la fiecare trecere a algoritmului Bellman-Ford, relaxarea muchiilor este realizată în ordinea descrisă în continuare. Înainte de prima trecere, atribuim o ordine liniară arbitrară $v_1, v_2, ..., v_{|V|}$ vârfurilor grafului de intrare $G = (V, E)$. În continuare, partiţionăm mulţimea de muchii E în $E_f \cup E_b$ unde $E_f = \{(v_i, v_j) \in E : i < j\}$ şi $E_b = \{(v_i v_j) \in E : i > j\}$. Definim $G_f = (V, E_f)$ şi $G_b = (V, E_b)$.

a. Demonstraţi că G_f este graf aciclic cu ordinea topologică $\langle v_1, v_2, ..., v_{|V|}\rangle$ şi G_b este aciclic cu ordinea topologică $\langle v_{|V|}, v_{|V|-1}, ..., v_1\rangle$.

Să presupunem că dorim să implementăm fiecare trecere a algoritmului Bellman-Ford în modul următor. Vizităm vârfurile în ordinea $v_1, v_2, ..., v_{|V|}$, relaxând muchiile mulţimii E_f. Apoi vizităm fiecare vârf în ordinea $v_{|V|}, v_{|V|-1}, \ldots, v_1$, relaxând fiecare muchie a mulţimii E_b care pleacă din vârf.

b. Demonstraţi pentru schema descrisă, că dacă G nu conţine cicluri de cost negativ accesibile din vârful sursă s, atunci după numai $\lceil |V|/2 \rceil$ treceri prin mulţimea muchiilor, avem $d[v] = \delta(s, v)$ pentru orice vârf $v \in V$.

c. Cum se modifică, în acest caz, timpul de execuţie al algoritmului Bellman-Ford?

25-2 Imbricarea cutiilor
Spunem că o cutie d-dimensională, având dimensiunile $(x_1, x_2, ..., x_d)$ este ***imbricată*** într-o a doua cutie de dimensiuni $(y_1, y_2, ..., y_d)$ dacă există o permutare π a mulţimii $\{1, 2, ..., d\}$ astfel încât, $x_{\pi(1)} < y_1, x_{\pi(2)} < y_2, ..., x_{\pi(d)} < y_d$.

a. Justificaţi că relaţia de imbricare este tranzitivă.

b. Propuneţi o metodă eficientă pentru a decide dacă o cutie d-dimensională este imbricată sau nu într-o a două cutie.

c. Presupunem că este dată o mulţime de cutii d-dimensionale, $\{B_1, B_2, ..., B_n\}$. Descrieţi un algoritm eficient pentru determinarea celei mai lungi secvenţe de cutii $\{B_{i_1}, B_{i_2}, ..., B_{i_k}\}$ cu proprietatea că B_{i_j} este imbricată în $B_{i_{j+1}}$ pentru $j = 1, 2, ..., k-1$. Evaluaţi timpul de execuţie al algoritmului propus în termenii valorilor n şi d.

25-3 Arbitraj

Prin ***arbitraj*** înţelegem utilizarea discrepanţelor existente între ratele de schimb pentru transformarea unei unităţi dintr-un anumit tip de valută în una sau mai multe unităţi de acelaşi tip. De exemplu, să presupunem că 1 dolar american poate cumpăra $0,7$ lire englezeşti; 1 liră englezească poate cumpăra $9,5$ franci francezi şi 1 franc francez poate cumpăra $0,16$ dolari americani. Atunci, prin conversia descrisă, cu 1 dolar american se pot cumpăra $0,7 \times 9,5 \times 0,16 = 1,064$ dolari americani, deci se obţine un profit de $6,4$ procente.

Presupunem că sunt date n tipuri de valută $c_1, c_2, ..., c_n$ şi un tabel R de dimensiune $n \times n$ a ratelor de schimb astfel încât o unitate din tipul c_i poate cumpăra $R[i, j]$ unităţi din tipul c_j

a. Propuneţi un algoritm eficient pentru a decide dacă există sau nu o secvenţă $\langle c_{i_1}, c_{i_2}, ..., c_{i_k} \rangle$ astfel încât

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Analizaţi timpul de execuţie al algoritmului propus.

b. Să se propuna un algoritm eficient pentru tipărirea unei astfel de secvenţe în cazul în care există. Analizaţi timpul de execuţie al algoritmului propus.

25-4 Algoritmul de scalare Gabow pentru drumurile minime de sursă unică

Un algoritm de ***scalare*** rezolvă o problemă prin considerarea în momentul iniţial numai a bitului de ordin maxim din fiecare dată de intrare relevantă (cum este, de exemplu, costul unei muchii). Soluţia iniţială este rafinată prin luarea în considerare a bitului de ordin imediat următor, în continuare procesul fiind iterat, la fiecare etapă fiind efectuată o nouă rafinare a soluţiei, până când toţi biţii sunt utilizaţi şi s-a calculat soluţia corectă.

În această problemă, ne interesează un algoritm pentru determinarea drumurilor minime de sursă unică prin scalarea costurilor muchiilor. Presupunem că avem un graf orientat $G = (V, E)$, ale cărui muchii au drept costuri numere întregi nenegative w. Fie $W = \max_{(u,v) \in E}\{w(u, v)\}$. Ne propunem să obţinem un algoritm cu timpul de execuţie $O(E \lg W)$. Presupunem că toate vârfurile pot fi atinse din sursă.

Algoritmul elimină, unul câte unul, biţii din reprezentarea binară corespunzătoare costurilor muchiilor, începând cu bitul cel mai semnificativ şi terminând cu cel mai puţin semnificativ. Mai exact, fie $k = \lceil \lg(W + 1) \rceil$ numărul de biţi din reprezentarea binară a lui W şi pentru $i = 1, 2, ..., k$, fie $w_i(u, v) = \lfloor w(u, v)/2^{k-i} \rfloor$. Cu alte cuvinte, $w_i(u, v)$ este versiunea "scalată cu un factor negativ" a lui $w(u, v)$ dacă i este cel mai semnificativ bit al lui $w(u, v)$. (Rezultă astfel, $w_k(u, v) = w(u, v)$ pentru toate muchiile $(u, v) \in E$.) De exemplu, dacă $k = 5$ şi $w(u, v) = 25$, a cărui reprezentare binară este $\langle 11001 \rangle$, atunci $w_3(u, v) = \langle 110 \rangle = 6$. De asemenea, dacă $k = 5$ şi $w(u, v) = \langle 00100 \rangle = 4$, atunci $w_3(u, v) = \langle 001 \rangle = 1$. Definim $\delta_i(u, v)$ costul unui drum minim de la vârful u la vârful v utilizând funcţia de cost w_i. Rezultă că pentru toate vârfurile $u, v \in V$,

$\delta_k(u,v) = \delta(u,v)$. Pentru vârful sursă s dat, algoritmul de scalare calculează mai întâi costurile drumurilor minime $\delta_1(s,v)$, pentru orice $v \in V$, în continuare calculează $\delta_2(s,v)$, pentru orice $v \in V$ şi aşa mai departe până se ajunge la $\delta_k(s,v)$, pentru orice $v \in V$. Vom presupune în cele ce urmează că $|E| \geq |V| - 1$ şi vom demonstra că evaluarea lui δ_i pe baza lui δ_{i-1} necesită un timp $O(E)$, deci acest algoritm se execută într-un timp $O(kE) = O(E \lg W)$.

a. Să presupunem că pentru toate vârfurile $v \in V$ avem $\delta(s,v) \leq |E|$. Demonstraţi că timpul necesar pentru calculul valorilor $\delta(s,v)$ pentru orice $v \in V$, este $O(E)$.

b. Demonstraţi că putem calcula $\delta_1(s,v)$ pentru orice $v \in V$, într-un timp $O(E)$. Ne preocupăm în continuare de calculul lui δ_i pe baza lui δ_{i-1}.

c. Demonstraţi că pentru $i = 2,3,...,k$, sau $w_i(u,v) = 2w_{i-1}(u,v)$ sau $w_i(u,v) = 2w_{i-1}(u,v) + 1$. Demonstraţi că pentru orice $v \in V$,

$$2\delta_{i-1}(s,v) \leq \delta_i(s,v) \leq 2\delta_{i-1}(s,v) + |V| - 1.$$

d. Pentru $i = 2,3,...,k$ şi fiecare muchie $(u,v) \in E$, definim

$$\widehat{w}_i(u,v) = w_i(u,v) + 2\delta_{i-1}(s,u) - 2\delta_{i-1}(s,v).$$

Demonstraţi că pentru $i = 2,3,...,k$ şi pentru toate vârfurile $u,v \in V$, $\widehat{w}_i(u,v)$ este un număr întreg nenegativ.

e. Fie $\widehat{\delta}_i(s,v)$ costul drumului minim de la s la v în cazul utilizării funcţiei de cost $\widehat{w}_i$. Demonstraţi că pentru $i = 2,3,...,k$ şi pentru toate vârfurile $v \in V$,

$$\delta_i(s,v) = \widehat{\delta}_i(s,v) + 2\delta_{i-1}(s,v)$$

şi că $\widehat{\delta}_i(s,v) \leq |E|$.

f. Arătaţi modul în care valorile $\delta_i(s,v)$ se pot calcula pe baza valorilor $\delta_{i-1}(s,v)$ pentru toate vârfurile $v \in V$, în timp $O(E)$ şi obţineţi că valorile $\delta(s,v)$ pentru orice $v \in V$ pot fi calculate în timp $O(E \lg W)$.

25-5 Algoritmul Karp pentru determinarea ciclului de cost mediu minim

Fie $G = (V,E)$ un graf orientat având funcţia de cost $w : E \to \mathbb{R}$ şi fie $n = |V|$. ***Costul mediu*** al unui ciclu $c = \langle e_1, e_2, ..., e_k \rangle$ format din muchii din E este definit prin

$$\mu(c) = \frac{1}{k} \sum_{i=1}^{k} w(e_i).$$

Fie $\mu^* = \min_c \mu(c)$ unde minimul este considerat peste mulţimea tuturor ciclurilor orientate din G. Spunem că ciclul c este un ***ciclu de cost mediu minim***, dacă $\mu(c) = \mu^*$. În cadrul acestei probleme ne interesează obţinerea unui algoritm eficient pentru calculul lui μ^*.

Putem presupune, fără restrângerea generalităţii, că orice vârf $v \in V$ este accesibil dintr-un vârf sursă $s \in V$. Fie $\delta(s,v)$ costul unui drum minim de la s la v şi fie $\delta_k(s,v)$ costul unui drum minim de la s la v constând din exact k muchii. Dacă nu există nici un drum de la s la v constând din exact k muchii, atunci $\delta_k(s,v) = \infty$.

a. Arătaţi că dacă $\mu^* = 0$, atunci G nu conţine cicluri de cost negativ şi pentru toate vârfurile $v \in V$, $\delta(s, v) = \min_{0 \le k \le n-1} \delta_k(s, v)$.

b. Arătaţi că dacă $\mu^* = 0$, atunci, pentru toate vârfurile $v \in V$,

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \ge 0.$$

(*Indicaţie:* Utilizaţi ambele proprietăţi de la punctul (***a***).)

c. Fie c un ciclu de cost 0 şi fie u, v două vârfuri din c. Fie x costul drumului de la u la v de-a lungul ciclului. Demonstraţi că $\delta(s, v) = \delta(s, u) + x$. (*Indicaţie:* Costul drumului de la v la u de-a lungul ciclului este $-x$.)

d. Arătaţi că dacă $\mu^* = 0$, atunci există un vârf v pe ciclul de cost mediu minim, astfel încât

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

(*Indicaţie:* Arătaţi că un drum minim către orice vârf aflat pe ciclul de cost mediu minim poate fi extins de-a lungul ciclului pentru a rezulta un drum minim la următorul vârf în acest ciclu.)

e. Arătaţi că dacă $\mu^* = 0$, atunci

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0.$$

f. Arătaţi că dacă adunăm o constantă t la costul fiecărei muchii din G, atunci valoarea lui μ^* este incrementată cu t. Utilizând această proprietate demonstraţi că

$$\mu^* = \min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k}.$$

g. Propuneţi un algoritm de timp $O(VE)$ pentru calculul lui μ^*.

Note bibliografice

Algoritmul Dijkstra [55] a apărut în 1959, dar în varianta iniţială nu a fost menţionată coada de priorităţi. Algoritmul Bellman-Ford este bazat pe doi algoritmi distincţi şi anume Bellman [22] şi Ford [71]. Bellman descrie relaţia dintre drumurile minime şi constrângerile cu diferenţe. Lawler [132] descrie algoritmul în timp liniar pentru drumurile minime într-un graf orientat aciclic pe care îl consideră în general cunoscut.

Când costurile muchiilor sunt numere întregi relativ mici, rezolvarea problemei drumurilor minime de sursă unică poate fi realizată pe baza unor algoritmi mai eficienţi. Ahuja, Mehlhorn, Orlin şi Tarjan [6] au propus un algoritm de complexitate temporală $O(E + V\sqrt{\lg W})$ pentru grafuri având costurile muchiilor numere nenegative, unde W este valoarea maximă a costurilor

muchiilor grafului. De asemenea, ei au propus un algoritm uşor de programat de timp $O(E + V \lg W)$. Pentru grafuri având costurile muchiilor numere nenegative, algoritmul propus de către Gabow şi Tarjan [77] se execută într-un timp $O(\sqrt{V}E \lg(VW))$ unde W este valoarea maximă a costurilor muchiilor grafului.

Papadimitriou şi Steiglitz [154] prezintă consideraţii interesante asupra metodei simplex şi algoritmul elipsoidului ca şi asupra mai multor algoritmi corelaţi cu programarea liniară. Algoritmul simplex pentru programarea liniară a fost inventat de G. Danzig în 1947. Diferitele variante ale algoritmului simplex sunt şi astăzi cele mai frecvent utilizate metode pentru rezolvarea problemelor de programare liniară. Algoritmul elipsoidului este datorat lui L. G. Khachian care l-a introdus în 1979 şi este bazată pe o lucrare mai veche publicată de N. Z. Shor, D. B. Judin şi A. S. Nemirovskii. Algoritmul Karmarkar este descris în [115].

26 Drumuri minime între toate perechile de vârfuri

În acest capitol vom studia problema determinării drumurilor minime dintre toate perechile de vârfuri ale unui graf. Această problemă poate apărea dacă dorim să creăm un tabel al distanţelor dintre toate perechile de oraşe care sunt desenate pe o hartă. La fel ca în capitolul 25, se dă un graf orientat, cu costuri $G = (V, E)$, cu o funcţie de cost $w : E \to \mathbb{R}$, care pune în corespondenţă arcele cu anumite costuri care sunt valori reale. Dorim să găsim, pentru fiecare pereche de vârfuri $u, v \in V$, un drum de cost minim de la u la v, unde costul drumului este suma costurilor arcelor care formează acel drum. De obicei, dorim ca ieşirea să apară sub forma unui tabel: elementul din linia u şi coloana v ar trebui să conţină costul drumului minim de la u la v.

Putem rezolva problema determinării drumurilor minime între toate perechile de vârfuri, rulând, pentru fiecare vârf din $|V|$, un algoritm de găsire a drumului minim de la acel vârf la toate celelalte. Dacă toate costurile arcelor sunt nenegative, putem folosi algoritmul lui Dijkstra. Dacă folosim implementarea unei cozi de prioritate sub formă de vector, timpul de execuţie este $O(V^3 + VE) = O(V^3)$. Implementarea cozii de prioritate, folosind heap-uri binare, duce la un timp de execuţie de ordinul $O(VE \lg V)$, ceea ce reprezintă o îmbunătăţire dacă graful este rar. Dacă implementăm coada de prioritate folosind un heap Fibonacci, atunci timpul de execuţie va fi $O(V^2 \lg V + VE)$.

Dacă sunt permise arce având costuri negative nu mai putem folosi algoritmul lui Dijkstra. În locul acestuia, putem folosi algoritmul Bellman-Ford pentru fiecare vârf. Timpul de execuţie rezultat este $O(V^2E)$, adică $O(V^4)$ pe un graf dens. În acest capitol vom vedea cum putem implementa algoritmi mai eficienţi. De asemenea, vom cerceta relaţia dintre problema drumurilor minime între toate perechile de vârfuri şi înmulţirea matricelor şi îi vom cerceta structura algebrică.

Spre deosebire de algoritmii de drum minim de sursă unică care folosesc reprezentarea grafului ca listă de adiacenţă, majoritatea algoritmilor din acest capitol vor folosi reprezentarea prin matrice de adiacenţă. (Algoritmul lui Johnson pentru grafuri rare foloseşte liste de adiacenţă.) Intrarea este o matrice W, de dimensiune $n \times n$, reprezentând costurile arcelor unui graf orientat $G = (V, E)$ având n vârfuri. Adică $W = (w_{ij})$ unde

$$w_{ij} = \begin{cases} 0 & \text{dacă } i = j, \\ \text{costul arcului}(i,j) & \text{dacă } i \neq j \text{ şi } (i,j) \in E, \\ \infty & \text{dacă } i \neq j \text{ şi } (i,j) \notin E. \end{cases} \tag{26.1}$$

Sunt permise arce cu costuri negative, dar, pentru moment, presupunem că graful de intrare nu conţine cicluri de cost negativ.

Ieşirea sub formă de tabel a algoritmilor de găsire a drumurilor minime între toate perechile de vârfuri este o matrice $D = (d_{ij})$ de dimensiuni $n \times n$, ale cărei elemente reprezintă costul drumului minim de la i la j. Aceasta înseamnă că, dacă notăm cu $\delta(i, j)$ costul minim al drumului de la i la j (la fel ca în capitolul 25), atunci, la final, vom avea $d_{ij} = \delta(i, j)$.

Pentru a rezolva problema găsirii drumurilor de cost minim între toate perechile de vârfuri, trebuie să calculăm, pe lângă costurile drumurilor minime, şi ***matricea predecesorilor*** $\Pi = (\pi_{ij})$, unde π_{ij} este NIL în cazul în care $i = j$ sau nu există drum de la i la j. În celelalte

cazuri, elementul π_{ij} este predecesorul lui j într-un drum minim de la i. La fel cum subgraful predecesorilor G_π din capitolul 25 este un arbore al drumurilor minime, subgraful produs de a i-a linie a matricei Π ar trebui să fie un arbore al drumurilor minime cu rădăcina i. Pentru fiecare vârf $i \in V$, definim ***subgraful predecesorilor*** al lui G pentru vârful i ca $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, unde

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\} \text{ şi } E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} \text{ şi } \pi_{ij} \neq \text{NIL}\}.$$

Dacă $G_{\pi,i}$ este un arbore de drumuri minime, atunci următoarea procedură, care este o versiune modificată a procedurii TIPĂREŞTE-DRUM din capitolul 23, tipăreşte un drum minim de la vârful i la vârful j.

TIPĂREŞTE-DRUMURILE-MINIME-DINTRE-TOATE-PERECHILE(Π, i, j)
1: **dacă** $i = j$ **atunci**
2: tipăreşte i
3: **altfel**
4: **dacă** π_{ij} =NIL **atunci**
5: tipăreşte "Nu există drum de la" i "la" j
6: **altfel**
7: TIPĂREŞTE-DRUMURILE-MINIME-DINTRE-TOATE-PERECHILE (Π, i, π_{ij})
8: tipăreşte j

Pentru a scoate în evidenţă trăsăturile esenţiale ale algoritmilor, pentru toate perechile din acest capitol, nu vom studia detaliat crearea şi proprietăţile matricei predecesorilor aşa cum am făcut pentru subgrafurile predecesorilor din capitolul 25. Elementele de bază sunt acoperite de câteva dintre exerciţii.

Rezumatul capitolului

În secţiunea 26.1 este prezentat un algoritm de programare dinamică, bazat pe înmulţirea matricelor pentru a rezolva problema găsirii drumurilor minime dintre toate perechile de drumuri. Folosind tehnica "ridicării repetate la pătrat", acest algoritm poate fi făcut să ruleze în timp $\Theta(V^3 \lg V)$. Un alt algoritm de programare dinamică, algoritmul Floyd-Warshall, rulează în timp $\Theta(V^3)$. În secţiunea 26.2 se acoperă, de asemenea, problema găsirii închiderii tranzitive a unui graf orientat care are legătură cu problema găsirii drumurilor minime dintre toate perechile de vârfuri. Algoritmul lui Johnson este prezentat în secţiunea 26.3. Spre deosebire de alţi algoritmi prezentaţi în acest capitol, algoritmul lui Johnson foloseşte reprezentarea grafurilor ca liste de adiacenţă. Acest algoritm rezolvă problema în timp $O(V^2 \lg V + VE)$, ceea ce îl face eficient pentru grafuri mari, rare. În final, în secţiunea 26.4 vom studia o structură algebrică numită "semiinel închis" care permite multor algoritmi de drum minim să fie aplicaţi ca gazde pentru alte probleme pentru toate perechile care nu implică drumuri minime.

Înainte de a merge mai departe, trebuie să stabilim câteva convenţii pentru reprezentarea prin matrice de adiacenţă. În primul rând, vom presupune că graful de intrare $G = (V, E)$ are n vârfuri, deci $n = |V|$. În al doilea rând, vom nota matricele cu majusculă: W, D, şi elementele lor cu litere mici şi indici inferiori: w_{ij}, d_{ij}. Unele matrice pot avea indici superiori între paranteze pentru a indica iteraţiile: $D^{(m)} = (d_{ij}^{(m)})$. Pentru o matrice A de dimensiuni $n \times n$, vom presupune că atributul $linii[A]$ conţine valoarea n.

26.1. Drumuri minime şi înmulţirea matricelor

În această secţiune este prezentat un algoritm de programare dinamică pentru problema drumurilor minime dintre toate perechile de vârfuri pe un graf orientat $G = (V, E)$. Fiecare buclă majoră a programului dinamic va apela o operaţie care este similară înmulţirii matricelor, astfel încât acest algoritm va arăta ca o înmulţire repetată de matrice. Vom începe prin proiectarea unui algoritm în timp $\Theta(V^4)$ pentru problema drumurilor minime dintre toate perechile de vârfuri şi vom îmbunătăţi acest algoritm, astfel încât să ruleze în timp $\Theta(V^3 \lg V)$.

Înainte de a merge mai departe, să recapitulăm paşii din capitolul 16 pentru dezvoltarea unui algoritm de programare dinamică.

1. Se caracterizează structura unei soluţii optime.
2. Se defineşte recursiv valoarea unei soluţii optime.
3. Se calculează valoarea soluţiei optime într-o manieră bottom-up.

(Al patrulea pas, obţinerea unei soluţii optime din informaţiile deja calculate, este tratat în cadrul exerciţiilor.)

Structura unui drum minim

Începem prin a caracteriza structura unei soluţii optime. Pentru problema drumurilor minime pentru toate perechile dintr-un graf $G = (V, E)$, am demonstrat (lema 25.1) că toate subdrumurile unui drum minim sunt drumuri minime. Să presupunem că graful este reprezentat printr-o matrice de adiacenţă $W = (w_{ij})$. Să considerăm un drum minim p de la vârful i la vârful j şi să presupunem că p conţine cel mult m arce. Presupunând că nu există cicluri de cost negativ, m este finit. Dacă $i = j$, atunci p are costul 0 şi nu conţine nici un arc. Dacă vârfurile i şi j sunt distincte, atunci descompunem drumul p în $i \overset{p'}{\rightsquigarrow} k \rightarrow j$, unde drumul p' conţine, acum, cel mult $m-1$ arce. Mai mult, conform lemei 25.1, p' este un drum minim de la i la k. În concluzie, datorită corolarului 25.2, avem $\delta(i, j) = \delta(i, k) + w_{kj}$.

O soluţie recursivă pentru problema drumurilor minime dintre toate perechile

Fie $d_{ij}^{(m)}$ costul minim al drumului de la i la j care conţine cel mult m arce. Când $m = 0$, există un drum minim de la i la j care nu conţine nici o muchie, dacă şi numai dacă $i = j$. În concluzie,

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{dacă } i = j, \\ \infty & \text{dacă } i \neq j. \end{cases}$$

Pentru $m \geq 1$, calculăm $d_{ij}^{(m)}$ ca fiind minimul dintre $d_{ij}^{(m-1)}$ (costul celui mai scurt drum de la i la j care conţine cel mult $m-1$ arce) şi costul minim al oricărui drum de la i la j care conţine

cel mult m arce şi este obţinut luând în considerare toţi predecesorii k ai lui j. Deci putem defini recursiv

$$d_{ij}^{(m)} = \min\left(d_{ij}^{(m-1)}, \min_{1\le k\le n}\left\{d_{ik}^{(m-1)} + w_{kj}\right\}\right) = \min_{1\le k\le n}\left\{d_{ik}^{(m-1)} + w_{kj}\right\}. \tag{26.2}$$

Ultima egalitate rezultă din faptul că $w_{jj} = 0$ oricare ar fi j.

Care sunt costurile $\delta(i, j)$ ale drumurilor minime? Dacă graful nu conţine nici un ciclu de cost negativ, atunci toate drumurile minime sunt elementare şi conţin, cel mult, $n - 1$ arce. Un drum de la vârful i la vârful j cu mai mult de $n - 1$ arce nu poate avea un cost mai mic decât cel mai scurt drum de la i la j. În concluzie, valorile costurilor drumurilor minime sunt date de:

$$\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \cdots. \tag{26.3}$$

Determinarea ascendentă a costurilor drumurilor minime

Considerând ca intrare matricea $W = (w_{ij})$, vom determina o serie de matrice $D^{(1)}$, $D^{(2)}$, ..., $D^{(n-1)}$, unde, pentru $m = 1, 2, \ldots, n-1$, avem $D^{(m)} = \left(d_{ij}^{(m)}\right)$. Matricea finală $D^{(n-1)}$ va conţine costurile drumurilor minime. Observaţi că, deoarece $d_{ij}^{(1)} = w_{ij}$ pentru toate vârfurile $i, j \in V$, avem $D^{(1)} = W$.

Esenţa algoritmului constă în următoarea procedură care, dându-se matricele $D^{(m-1)}$ şi W, returnează matricea $D^{(m)}$. Cu alte cuvinte, extinde drumurile minime determinate anterior cu încă un arc.

EXTINDE-DRUMURI-MINIME(D, W)
1: $n \leftarrow linii[D]$
2: fie $D' = (d'_{ij})$ o matrice de dimensiuni $n \times n$
3: **pentru** $i \leftarrow 1, n$ **execută**
4: **pentru** $j \leftarrow 1, n$ **execută**
5: $d'_{ij} \leftarrow \infty$
6: **pentru** $k \leftarrow 1, n$ **execută**
7: $d'_{ij} \leftarrow \min(d'_{ij}, d_{ik} + w_{kj})$
8: **returnează** D'

Procedura determină matricea $D' = (d'_{ij})$ pe care o returnează la sfârşit. Acest lucru este realizat calculând ecuaţia (26.2) pentru orice i şi j, folosind D pentru $D^{(m-1)}$ şi D' pentru $D^{(m)}$. (Procedura este scrisă fără indici superiori pentru ca matricele de intrare şi de ieşire să fie independente de m.) Timpul de execuţie al procedurii este $\Theta(n^3)$ datorită celor trei bucle **pentru** imbricate.

Putem observa, acum, legătura cu înmulţirea matricelor. Să presupunem că dorim să calculăm produsul $C = A \cdot B$ a două matrice A şi B de dimensiuni $n \times n$. Atunci, pentru $i, j = 1, 2, \ldots, n$, vom calcula

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}. \tag{26.4}$$

Observaţi că, dacă în ecuaţia (26.2) vom face substituţiile

$$d^{(m-1)} \to a, w \to b, d^{(m)} \to c, \min \to +, + \to \cdot$$

vom obţine ecuaţia (26.4). Ca urmare, dacă efectuăm aceste schimbări în EXTINDE-DRUMURI-MINIME şi înlocuim ∞ (elementul neutru pentru min) cu 0 (elementul neutru pentru +), obţinem, direct procedura în timp $\Theta(n^3)$ pentru înmulţirea matricelor.

ÎNMULŢIRE-MATRICE(A, B)
1: $n \leftarrow linii[A]$
2: fie C o matrice de dimensiuni $n \times n$
3: **pentru** $i \leftarrow 1, n$ **execută**
4: **pentru** $j \leftarrow 1, n$ **execută**
5: $c_{ij} \leftarrow 0$
6: **pentru** $k \leftarrow 1, n$ **execută**
7: $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$
8: **returnează** C

Întorcându-ne la problema drumurilor minime între toate perechile, determinăm costurile drumurilor minime extinzând drumurile minime arc cu arc. Notând cu $A \cdot B$ matricea "produs" returnată de EXTINDE-DRUMURI-MINIME(A, B), determinăm şirul de $n - 1$ matrice

$$\begin{aligned} D^{(1)} &= D^{(0)} \cdot W &&= W, \\ D^{(2)} &= D^{(1)} \cdot W &&= W^2, \\ D^{(3)} &= D^{(2)} \cdot W &&= W^3, \\ &\vdots \\ D^{(n-1)} &= D^{(n-2)} \cdot W &&= W^{n-1}. \end{aligned}$$

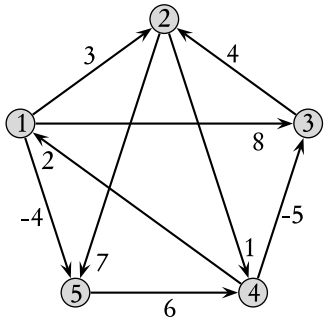
După cum am arătat anterior, matricea $D^{(n-1)} = W^{n-1}$ conţine costurile drumurilor minime. Următoarea procedură determină acest şir în timp $\Theta(n^4)$.

DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-LENT(W)
1: $n \leftarrow linii[W]$
2: $D^{(1)} \leftarrow W$
3: **pentru** $m \leftarrow 2, n - 1$ **execută**
4: $D^{(m)} \leftarrow$ EXTINDE-DRUMURI-MINIME$(D^{(m-1)}, W)$
5: **returnează** $D^{(n-1)}$

În figura 26.1 se poate vedea un graf şi matricele $D^{(m)}$ determinate de procedura DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-LENT.

Îmbunătăţirea timpului de execuţie

Scopul nostru nu este determinarea *tuturor* matricelor $D^{(m)}$: ne interesează numai matricea $D^{(n-1)}$. Să ne reamintim că, în absenţa ciclurilor cu cost negativ, ecuaţia (26.3) implică $D^{(m)} = D^{(n-1)}$ pentru toate numerele întregi $m \geq n - 1$. Putem determina $D^{(n-1)}$ cu ajutorul a numai $\lceil \lg(n-1) \rceil$ produse de matrice determinând şirul:



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -4 \\ 7 & 4 & 0 & 5 & 1 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figura 26.1 Un graf orientat şi şirul de matrice $D^{(m)}$ determinat de DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-LENT. Cititorul poate verifica faptul că $D^{(5)} = D^{(4)} \cdot W$ este egală cu $D^{(4)}$, deci $D^{(m)} = D^{(4)}$ pentru orice $m \geq 4$.

$$\begin{aligned} D^{(1)} &= W, \\ D^{(2)} &= W^2 &&= W \cdot W, \\ D^{(4)} &= W^4 &&= W^2 \cdot W^2, \\ D^{(8)} &= W^8 &&= W^4 \cdot W^4, \\ &\vdots \\ D^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}. \end{aligned}$$

Deoarece $2^{\lceil \lg(n-1) \rceil} \geq n-1$, produsul final $D^{(2^{\lceil \lg(n-1) \rceil})}$ este egal cu $D^{(n-1)}$.

Următoarea procedură determină şirul anterior de matrice folosind o tehnică numită ***ridicare repetată la pătrat***.

DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-MAI-RAPID(W)
1: $n \leftarrow linii[W]$
2: $D^{(1)} \leftarrow W$
3: $m \leftarrow 1$
4: **cât timp** $n - 1 > m$ **execută**
5: $\quad D^{(2m)} \leftarrow$ EXTINDE-DRUMURI-MINIME($D^{(m)}, D^{(m)}$)

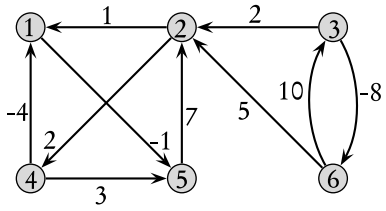


Figura 26.2 Un graf orientat cu costuri pentru exerciţiile 26.1-1, 26.2-1 şi 26.3-1.

6: $m \leftarrow 2m$
7: **returnează** $D^{(m)}$

La fiecare iteraţie a buclei **cât timp** din liniile 4–6, calculăm $D^{(2m)} = \left(D^{(m)}\right)^2$, începând cu $m = 1$. La sfârşitul fiecărei iteraţii, dublăm valoarea lui m. În ultima iteraţie este determinată $D^{(n-1)}$ calculându-se, de fapt, $D^{(2m)}$ pentru un anumit $n-1 \leq 2m < 2n-2$. Din ecuaţia (26.3), rezultă că $D^{(2m)} = D^{(n-1)}$. La următoarea testare a condiţiei din linia 4, m a fost dublat, deci vom avea $n-1 \leq m$, aşadar condiţia nu va fi adevărată şi procedura va returna ultima matrice determinată.

Timpul de execuţie al procedurii DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-MAI-RAPID este $\Theta(n^3 \lg n)$ deoarece fiecare dintre cele $\lceil \lg(n-1) \rceil$ înmulţiri de matrice are nevoie de un timp $\Theta(n^3)$. Observaţi că avem un cod scurt care nu conţine structuri de date complicate, deci constanta ascunsă în notaţia Θ este mică.

Exerciţii

26.1-1 Executaţi algoritmul DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-LENT pe graful orientat cu costuri din figura 26.2 arătând care sunt matricele care rezultă pentru fiecare iteraţie a buclei respective. Realizaţi acelaşi lucru pentru algoritmul algoritmului DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-MAI-RAPID.

26.1-2 De ce este nevoie ca w_{ii} să fie egal cu 0 pentru orice $1 \leq i \leq n$?

26.1-3 Cărei matrice din înmulţirea obişnuită a matricelor îi corespunde matricea

$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

folosită în algoritmii de drum minim.

26.1-4 Arătaţi cum poate fi exprimată problema drumului minim de la un vârf la restul vârfurilor cu ajutorul unui produs de matrice şi un vector. Descrieţi modul în care calculul acestui produs corespunde unui algoritm de genul Bellman-Ford (vezi 25.3).

26.1-5 Să presupunem că dorim să determinăm şi vârfurile care compun drumurile minime în algoritmii din această secţiune. Arătaţi modul în care se determină, în timp $O(n^3)$, matricea predecesorilor Π cu ajutorul matricei D a costurilor drumurilor minime.

26.1-6 Vârfurile care compun drumurile minime pot fi determinate în acelaşi timp cu costurile drumurilor minime. Definim $\pi_{ij}^{(m)}$ ca fiind predecesorul vârfului j pe oricare drum de cost minim de la i la j care conţine cel mult m arce. Modificaţi EXTINDE-DRUMURI-MINIME şi DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-LENT pentru a determina matricele $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ pe măsură ce sunt determinate matricele $D^{(1)}, D^{(2)}, \ldots, D^{(n-1)}$.

26.1-7 Procedura DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-MAI-RAPID, aşa cum a fost scrisă, necesită stocarea a $\lceil \lg(n-1) \rceil$ matrice, fiecare având n^2 elemente, deci spaţiul necesar de memorie este $\Theta(n^2 \lg n)$. Modificaţi procedura pentru ca ea să necesite numai un spaţiu $\Theta(n^2)$ folosind doar două matrice de dimensiuni $n \times n$.

26.1-8 Modificaţi DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-MAI-RAPID pentru a detecta prezenţa unui ciclu de cost negativ.

26.1-9 Daţi un algoritm eficient pentru a găsi, într-un graf, lungimea (numărul de muchii) unui ciclu de lungime minimă al cărui cost este negativ.

26.2. Algoritmul Floyd-Warshall

În această secţiune vom folosi o formulare diferită a programării dinamice pentru a rezolva problema drumului minim între toate perechile pe un graf orientat $G = (V, E)$. Algoritmul rezultat, cunoscut sub numele de ***algoritmul Floyd-Warshall***, se execută în timp $\Theta(V^3)$. Ca şi anterior, pot apărea arce cu costuri negative, dar vom presupune că nu există cicluri de cost negativ. La fel ca în secţiunea 26.1, vom urma paşii programării dinamice pentru a dezvolta algoritmul. După ce vom studia algoritmul rezultat, vom prezenta o metodă similară pentru a găsi închiderea tranzitivă a unui graf orientat.

Structura unui drum minim

În algoritmul Floyd-Warshall folosim o caracterizare diferită a structurii unui drum minim faţă de cea folosită în algoritmii de drum minim bazaţi pe înmulţirea matricelor. Algoritmul ia în considerare vârfurile "intermediare" ale unui drum minim, unde un vârf ***intermediar*** al unui drum elementar $p = \langle v_1, v_2, \ldots, v_l \rangle$ este oricare vârf p diferit de v_1 şi de v_l, adică orice vârf din mulţimea $\{v_2, v_3, \ldots, v_{l-1}\}$.

Algoritmul Floyd-Warshall este bazat pe următoarea observaţie. Fie $V = \{1, 2, \ldots, n\}$ mulţimea vârfurilor lui G. Considerăm submulţimea $\{1, 2, \ldots, k\}$ pentru un anumit k. Pentru oricare pereche de vârfuri $i, j \in V$, considerăm toate drumurile de la i la j ale căror vârfuri intermediare fac parte din mulţimea $\{1, 2, \ldots, k\}$. Fie p drumul de cost minim dintre aceste drumuri. (Drumul p este elementar deoarece presupunem că G nu conţine cicluri de cost negativ.) Algoritmul Floyd-Warshall exploatează o relaţie între drumul p şi drumul minim de la i la j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$. Relaţia depinde de statutul lui k: acesta poate fi sau nu un vârf intermediar al lui p.

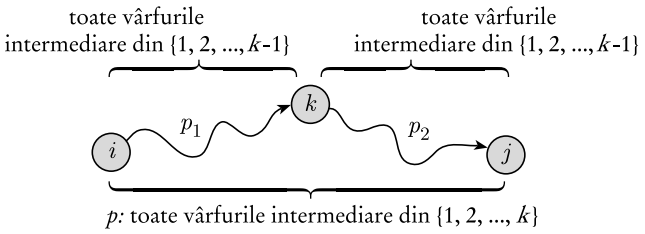


Figura 26.3 Drumul p este un drum minim de la vârful i la vârful j şi k este vârful intermediar al lui p cu numărul cel mai mare. Drumul p_1, porţiunea din drumul p de la vârful i la vârful k, are toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$. Aceasta este valabil şi pentru drumul p_2 de la vârful k la vârful j.

- Dacă k nu este un vârf intermediar al drumului p, atunci toate vârfurile intermediare ale drumului p fac parte din mulţimea $\{1, 2, \ldots, k-1\}$. Ca urmare, un drum minim de la vârful i la vârful j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$ este, de asemenea, un drum minim de la i la j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$.

- Dacă k este un vârf intermediar al drumului p, atunci vom împărţi p în $i \overset{p_1}{\rightsquigarrow} k \overset{p_2}{\rightsquigarrow} j$ după cum se poate vedea în figura 26.3. Din lema 25.1 rezultă că p_1 este drumul minim de la i la k cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$. De fapt, vârful k nu este un vârf intermediar al drumului p_1, deci p_1 este un drum minim de la i la k cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$. În mod similar, arătăm că p_2 este un drum minim de la vârful k la vârful j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$.

O soluţie recursivă la problema drumurilor minime între toate perechile de vârfuri

Bazându-ne pe observaţia anterioară, definim o formulare recursivă a estimărilor drumului minim faţă de cea din secţiunea 26.1. Fie $d_{ij}^{(k)}$ costul drumului minim de la vârful i la vârful j, cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$. Când $k = 0$, un drum de la vârful i la vârful j, cu nici un vârf intermediar al cărui număr este mai mare decât 0, nu are, de fapt, nici un vârf intermediar. În concluzie, acest drum conţine cel mult un arc, deci $d_{ij}^{(0)} = w_{ij}$. O definiţie recursivă este dată de:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{dacă } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{dacă } k \geq 1. \end{cases} \tag{26.5}$$

Matricea $D^{(n)} = \left(d_{ij}^{(n)}\right)$ conţine răspunsul final, $d_{ij}^{(n)} = \delta(i, j)$ pentru orice $i, j \in V$, deoarece toate vârfurile intermediare se află în mulţimea $\{1, 2, \ldots, n\}$.

Determinarea ascendentă a costurilor drumurilor minime

Următoarea procedură ascendentă este bazată pe recurenţa (26.5) şi poate fi folosită pentru a determina valorile $d_{ij}^{(k)}$ în ordinea valorilor crescătoare k. Intrarea ei este o matrice W de

dimensiuni $n \times n$ definită aşa cum se arată în ecuaţia (26.1). Procedura returnează matricea $D^{(n)}$ a costurilor drumurilor minime.

FLOYD-WARSHALL(W)
1: $n \leftarrow linii[W]$
2: $D^{(0)} \leftarrow W$
3: **pentru** $k \leftarrow 1, n$ **execută**
4: **pentru** $i \leftarrow 1, n$ **execută**
5: **pentru** $j \leftarrow 1, n$ **execută**
6: $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
7: **returnează** $D^{(n)}$

În figura 26.4 sunt prezentate matricele $D^{(k)}$ determinate de algoritmul Floyd-Warshall, aplicat grafului din figura 26.1.

Timpul de execuţie al algoritmului Floyd-Warshall este determinat de cele trei cicluri **pentru** imbricate din liniile 3–6. Fiecare execuţie a liniei 6 necesită un timp $O(1)$, deci algoritmul se execută în timp $\Theta(n^3)$. La fel ca în cazul ultimului algoritm din secţiunea 26.1, codul este scurt, fără structuri de date complicate, deci constanta ascunsă în notaţia Θ este mică. În concluzie, algoritmul Floyd-Warshall este destul de practic, chiar dacă grafurile de intrare au dimensiune medie.

Construirea unui drum minim

Există o varietate de metode diferite pentru construirea drumurilor minime în algoritmul Floyd-Warshall. O modalitate este de a determina matricea D a costurilor drumurilor minime şi de a construi, apoi, matricea Π a predecesorilor cu ajutorul matricei D. Această metodă poate fi implementată, astfel încât să ruleze în timp $O(n^3)$ (exerciţiul 26.1-5). Dându-se matricea Π a predecesorilor, procedura TIPĂREŞTE-DRUMURILE-MINIME-DINTRE-TOATE-PERECHILE poate fi folosită pentru a tipări vârfurile unui drum minim dat.

Putem determina matricea Π "în timp real" la fel cum algoritmul Floyd-Warshall determină matricea $D^{(k)}$. Mai exact, determinăm un şir de matrice $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, unde $\Pi = \Pi^{(n)}$ şi $\pi_{ij}^{(k)}$ este definit ca fiind predecesorul vârfului j în drumul minim de la vârful i cu toţi predecesorii în mulţimea $\{1, 2, \ldots, k\}$.

Putem da o formulă recursivă pentru $\pi_{ij}^{(k)}$. Când $k = 0$, un drum minim de la i la j nu are nici un vârf intermediar. Deci

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{dacă } i = j \text{ sau } w_{ij} = \infty, \\ i & \text{dacă } i \neq j \text{ sau } w_{ij} < \infty. \end{cases} \tag{26.6}$$

Pentru $k \geq 1$, dacă luăm în considerare drumul $i \rightsquigarrow k \rightsquigarrow j$, atunci predecesorul lui j pe care îl alegem este acelaşi cu predecesorul lui j pe care îl alegem pe un drum minim de la k cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k-1\}$. Altfel, alegem acelaşi predecesor al lui j pe care l-am ales într-un drum minim de la i, cu toate vârfurile intermediare din mulţimea $\{1, 2, \ldots, k-1\}$. Mai exact, pentru $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{dacă } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{dacă } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases} \tag{26.7}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figura 26.4 Şirurile de matrice $D^{(k)}$ şi $\Pi^{(k)}$ determinat de algoritmul Floyd-Warshall pentru graful din figura 26.1.

Lăsăm includerea determinării matricei $\Pi^{(k)}$ în cadrul procedurii FLOYD-WARSHALL pe seama exerciţiului 26.2-3. În figura 26.4 se arată şirul $\Pi^{(k)}$ pe care algoritmul rezultat îl determină pentru graful din figura 26.1. Exerciţiul vă cere, de asemenea, să demonstraţi că subgraful $G_{\pi,i}$ al predecesorilor este un arbore de drumuri minime cu rădăcina i. Un alt mod de a reconstitui drumurile minime se cere în exerciţiul 26.2-6.

Închiderea tranzitivă a unui graf orientat

Dându-se un graf orientat $G = (V, E)$ cu mulţimea vârfurilor $V = \{1, 2, \ldots, n\}$, am putea dori să aflăm dacă în G există un drum de la i la j pentru toate perechile $i, j \in V$. ***Închiderea tranzitivă*** a lui G este definită ca fiind graful $G^* = (V, E^*)$, unde

$$E^* = \{(i, j) : \text{ există un drum de la vârful } i \text{ la vârful } j \text{ în } G\}.$$

O modalitate de a determina închiderea tranzitivă a unui graf în timp $\Theta(n^3)$ este de a atribui costul 1 tuturor muchiilor din E şi a rula algoritmul Floyd-Warshall. Dacă există un drum de la vârful i la vârful j, obţinem $d_{ij} < n$. În caz contrar, obţinem $d_{ij} = \infty$.

Există o altă modalitate similară de a determina închiderea tranzitivă a lui G în timp $\Theta(n^3)$ care, în practică, poate economisi timp şi spaţiu. Această metodă implică înlocuirea operaţiilor logice $\wedge$ (ŞI logic) şi $\vee$ (SAU logic) cu operaţiile aritmetice min şi + în algoritmul Floyd-Warshall. Pentru $i, j, k = 1, 2, \ldots, n$ definim $t_{ij}^{(k)}$ ca fiind 1, dacă în graful G există un drum de la vârful i la vârful j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$, şi 0 în caz contrar. Construim închiderea tranzitivă $G^* = (V, E^*)$ adăugând arcul (i, j) la E^* dacă şi numai dacă $t_{ij}^{(n)} = 1$. O definiţie recursivă pentru $t_{ij}^{(k)}$, analogă recurenţei (26.5), este

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{dacă } i \neq j \text{ şi } (i, j) \notin E, \\ 1 & \text{dacă } i = j \text{ sau } (i, j) \in E, \end{cases}$$

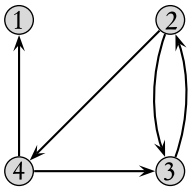
şi pentru $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right). \tag{26.8}$$

La fel ca în cazul algoritmului Floyd-Warshall, determinăm matricele $T^{(k)} = \left(t_{ij}^{(k)} \right)$ în ordinea crescătoare a lui k.

ÎNCHIDERE-TRANZITIVĂ(G)
1: $n \leftarrow |V[G]|$
2: **pentru** $i \leftarrow 1, n$ **execută**
3: **pentru** $j \leftarrow 1, n$ **execută**
4: **dacă** $i = j$ sau $(i, j) \in E[G]$ **atunci**
5: $t_{ij}^{(0)} \leftarrow 1$
6: **altfel**
7: $t_{ij}^{(0)} \leftarrow 0$
8: **pentru** $k \leftarrow 1, n$ **execută**
9: **pentru** $i \leftarrow 1, n$ **execută**
10: **pentru** $j \leftarrow 1, n$ **execută**
11: $t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right)$
12: **returnează** $T^{(n)}$

În figura 26.5 este prezentată matricea $T^{(k)}$ determinată de procedura ÎNCHIDERE-TRANZITIVĂ pe un graf dat. La fel ca în algoritmul Floyd-Warshall, timpul de execuţie pentru procedura ÎNCHIDERE-TRANZITIVĂ este $\Theta(n^3)$. Pe unele calculatoare operaţiile logice cu valori de dimensiune un bit sunt executate mai rapid decât operaţiile cu date de dimensiune un cuvânt.



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figura 26.5 Un graf orientat şi matricea $T^{(k)}$ calculată folosind algoritmul închiderii tranzitive.

Mai mult, deoarece algoritmul direct de determinare a închiderii tranzitive foloseşte numai valori logice şi nu valori întregi, spaţiul de memorie necesar este mai mic decât spaţiul necesar în cazul algoritmului Floyd-Warshall cu un factor corespunzător mărimii unui cuvânt de memorie.

În secţiunea 26.4 vom vedea că această corespondenţă între FLOYD-WARSHALL şi ÎNCHIDERE-TRANZITIVĂ este mai mult decât o simplă coincidenţă. Ambii algoritmi se bazează pe o structură algebrică numită "semiinel închis".

Exerciţii

26.2-1 Executaţi algoritmul Floyd-Warshall pe graful orientat cu costuri din figura 26.2. Precizaţi care este matricea $D^{(k)}$ care rezultă după fiecare iteraţie a buclei exterioare.

26.2-2 După cum am arătat anterior, algoritmul Floyd-Warshall necesită un spaţiu $\Theta(n^3)$ deoarece calculăm $d_{ij}^{(k)}$ pentru $i, j, k = 1, 2, \ldots, n$. Arătaţi că următoarea procedură, care pur şi simplu renunţă la indicii superiori, este corectă, deci este necesar doar un spaţiu $\Theta(n^2)$.

FLOYD-WARSHALL'(W)
1: $n \leftarrow linii[W]$
2: $D \leftarrow W$
3: **pentru** $k \leftarrow 1, n$ **execută**
4: **pentru** $i \leftarrow 1, n$ **execută**
5: **pentru** $j \leftarrow 1, n$ **execută**
6: $d_{ij} \leftarrow \min(d_{ij}, d_{ik} + d_{kj})$
7: **returnează** D

26.2-3 Modificaţi procedura FLOYD-WARSHALL incluzând determinarea matricelor $\Pi^{(k)}$ în conformitate cu ecuaţiile (26.6) şi (26.7). Demonstraţi riguros că, pentru orice $i \in V$, graful

$G_{\pi,i}$ al predecesorilor este un arbore de drumuri minime cu rădăcina i. (*Indicaţie:* Pentru a arăta că $G_{\pi,i}$ este aciclic arătaţi, mai întâi, că $\pi_{ij}^{(k)} = l$ implică $d_{ij}^{(k)} \geq d_{il}^{(k-1)} + w_{lj}$. Adaptaţi apoi demonstraţia lemei 25.8.)

26.2-4 Să presupunem că modificăm felul în care este tratată egalitatea în ecuaţia (26.7):

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{dacă } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{dacă } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Este corectă această definiţie alternativă a matricei Π a predecesorilor?

26.2-5 Cum poate fi folosită ieşirea algoritmului FLOYD-WARSHALL pentru a detecta prezenţa unui ciclu de cost negativ?

26.2-6 O altă modalitate de a reconstitui drumurile minime, în cazul algoritmului Floyd-Warshall, este folosirea valorilor $\phi_{ij}^{(k)}$ pentru $i, j, k = 1, 2, \ldots, n$, unde $\phi_{ij}^{(k)}$ este vârful intermediar din drumul de la i la j, numerotat cu cel mai mare număr. Daţi o definiţie recursivă pentru $\phi_{ij}^{(k)}$, modificaţi procedura FLOYD-WARSHALL pentru a determina valorile $\phi_{ij}^{(k)}$ şi rescrieţi TIPĂREŞTE-DRUMURILE-MINIME-ÎNTRE-TOATE-PERECHILE pentru a avea ca intrare matricea $\Phi = \left(\phi_{ij}^{(n)}\right)$. În ce măsură seamănă matricea Φ cu tabelul s din problema înmulţirii lanţului de matrice din secţiunea 16.1?

26.2-7 Daţi un algoritm de timp $O(VE)$ pentru determinarea închiderii tranzitive a unui graf orientat $G = (V, E)$.

26.2-8 Să presupunem că închiderea tranzitivă a unui graf orientat aciclic poate fi determinată într-un timp $f(|V|, |E|)$, unde f este monoton crescătoare care depinde $|V|$ şi $|E|$. Demonstraţi că timpul necesar determinării închiderii tranzitive $G^\star = (V, E^\star)$ a unui graf orientat obişnuit $G = (V, E)$ este $f(|V|, |E|) + O(V + E^\star)$.

26.3. Algoritmul lui Johnson pentru grafuri rare

Algoritmul lui Johnson găseşte drumurile minime între toate perechile de vârfuri, de timp $O(V^2 \lg V + VE)$; deci este asimptotic, mai bun şi decât ridicarea repetată la pătrat, şi decât algoritmul Floyd-Warshall pentru grafuri rare. Algoritmul returnează o matrice a costurilor drumurilor minime pentru toate perechile de vârfuri sau un mesaj care indică faptul că există cicluri de costuri negative. Algoritmul lui Johnson foloseşte ca subrutine atât algoritmul lui Dijkstra cât şi algoritmul Bellman-Ford care sunt descrise în capitolul 25.

Algoritmul lui Johnson foloseşte o tehnică de ***schimbare a costurilor*** care funcţionează după cum urmează. Dacă toate arcele grafului $G = (V, E)$ au costuri w nenegative, putem găsi drumurile minime dintre toate perechile de vârfuri, executând algoritmul lui Dijkstra pentru fiecare vârf: cu ajutorul cozii de prioritate implementate cu un heap Fibonacci, timpul de execuţie al acestui algoritm este $O(V^2 \lg V + VE)$. Dacă G are arce cu costuri negative determinăm o nouă mulţime de arce care au costuri nenegative, mulţime care ne permite să folosim aceeaşi metodă. Noua mulţime $\widehat{w}$ trebuie să satisfacă două proprietăţi importante.

1. Pentru toate perechile de vârfuri $u, v \in V$, un drum minim de la u la v, folosind funcţia de cost w este, de asemenea, un drum minim de la u la v folosind funcţia de cost $\widehat{w}$.
2. Pentru toate arcele (u, v), noul cost $\widehat{w}(u, v)$ este nenegativ.

După cum vom vedea foarte curând, preprocesarea lui G pentru a determina funcţia de cost $\widehat{w}$ poate fi efectuată într-un timp $O(VE)$.

Menţinerea drumurilor minime prin schimbarea costurilor

După cum arată următoarea lemă, este destul de uşor să facem o schimbare a costurilor care să respecte prima dintre proprietăţile anterioare. Notăm prin δ costurile drumurilor minime determinate pentru funcţia de cost w şi prin $\widehat{\delta}$ costurile drumurilor minime determinate pentru funcţia de cost $\widehat{w}$.

Lema 26.1 (Schimbarea costurilor nu schimbă drumurile minime) Dându-se un graf orientat cu costuri $G = (V, E)$ a cărui funcţie de cost este $w : E \to \mathbb{R}$, fie $h : V \to \mathbb{R}$ o funcţie care pune în corespondenţă vârfurile cu numere reale. Pentru fiecare arc $(u, v) \in E$), definim

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v). \tag{26.9}$$

Fie $p = \langle v_0, v_1, \ldots, v_k \rangle$ un drum de la vârful v_0 la vârful v_k. Atunci $w(p) = \delta(v_0, v_k)$ dacă şi numai dacă $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. De asemenea, G are cicluri de cost negativ folosind funcţia w dacă şi numai dacă are un ciclu de cost negativ, folosind funcţia $\widehat{w}$.

Demonstraţie. Începem prin a arăta că

$$\widehat{w}(p) = w(p) + h(v_0) - h(v_k). \tag{26.10}$$

Avem

$$\begin{aligned} \widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) = \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) = w(p) + h(v_0) - h(v_k). \end{aligned}$$

A treia egalitate rezultă în urma efectuării sumei din cea de-a doua egalitate.

Arătăm acum prin reducere la absurd că $w(p) = \delta(v_0, v_k)$ implică $\widehat{w}(p) = \widehat{\delta}(v_0, v_k)$. Să presupunem că există un drum mai scurt p' de la v_0 la v_k folosind funcţia de cost $\widehat{w}$. Atunci $\widehat{w}(p') < \widehat{w}(p)$. Din ecuaţia (26.10) rezultă că

$$w(p') + h(v_0) - h(v_k) = \widehat{w}(p') < \widehat{w}(p) = w(p) + h(v_0) - h(v_k),$$

ceea ce implică $w(p') < w(p)$. Însă aceasta contrazice ipoteza că p este drumul minim de la u la v folosind w. Demonstraţia reciprocei este similară.

În final, arătăm că G are cicluri de cost negativ folosind funcţia de cost w dacă şi numai dacă G are cicluri de cost negativ folosind funcţia de cost $\widehat{w}$. Să considerăm orice ciclu $c = \langle v_0, v_1, \ldots, v_k \rangle$, unde $v_0 = v_k$. Din ecuaţia (26.10) rezultă că

$$\widehat{w}(c) = w(c) + h(v_0) - h(v_k) = w(c),$$

deci c are cost negativ folosind w dacă şi numai dacă are cost negativ folosind $\widehat{w}$. ∎

Obţinerea costurilor nenegative prin schimbarea costurilor

Următorul scop este să ne asigurăm că este respectată a doua proprietate: vrem ca $\widehat{w}(u,v)$ să fie nenegativă pentru toate arcele $(u,v) \in E$. Dându-se un graf orientat cu costuri $G = (V,E)$ cu funcţia de cost $w : E \to \mathbb{R}$, construim un nou graf $G' = (V', E')$, unde $V' = V \cup \{s\}$ pentru un vârf nou $s \notin V$ şi $E' = E \cup \{(s,v) : s \in V\}$. Extindem funcţia de cost w, astfel încât $w(s,v) = 0$ oricare ar fi $v \in V$. Observaţi că, deoarece s nu are nici o muchie care intră în el, nu există drumuri minime în G' care îl conţin pe s în afară de cele care pornesc din s. Mai mult, G' nu are cicluri de cost negativ dacă şi numai dacă G nu are cicluri de cost negativ. În figura 26.6(a) este prezentat graful G' corespunzător grafului G din figura 26.1.

Să presupunem acum că G şi G' nu au cicluri de cost negativ. Definim $h(v) = \delta(s,v)$ pentru orice $v \in V'$. Din lema 25.3 rezultă că $h(v) \le h(u) + w(u,v)$ pentru toate arcele $(u,v) \in E'$. În concluzie, dacă definim noile costuri $\widehat{w}$ în concordanţă cu ecuaţia (26.9), avem $\widehat{w}(u,v) = w(u,v) + h(u) - h(v) \ge 0$, şi cea de-a doua proprietate este satisfăcută. În figura 26.6(b) este prezentat graful G' din figura 26.6(a) cu muchiile având costuri schimbate.

Determinarea drumurilor minime între toate perechile de vârfuri

Algoritmul lui Johnson pentru determinarea drumurilor minime dintre toate perechile de vârfuri foloseşte algoritmul Bellman-Ford (secţiunea 25.3) şi algoritmul lui Dijkstra (secţiunea 25.2) ca subrutine. Algoritmul presupune că muchiile sunt păstrate în liste de adiacenţă. Algoritmul returnează obişnuita matrice $D = d_{ij}$ de dimensiuni $|V| \times |V|$, unde $d_{ij} = \delta(i,j)$ sau un mesaj corespunzător pentru cazul în care graful conţine cicluri de cost negativ. (Pentru ca indicii matricei D să aibă sens, presupunem că vârfurile sunt numerotate de la 1 la $|V|$.)

JOHNSON(G)
1: determină G', unde $V[G'] = V[G] \cup \{s\}$ şi $E[G'] = E[G] \cup \{(s,v) : v \in V[G]\}$
2: **dacă** BELLMAN-FORD(G',w,s)=FALS **atunci**
3: tipăreşte "Graful de intrare conţine cel puţin un ciclu de cost negativ"
4: **altfel**
5: **pentru** fiecare vârf $v \in V[G']$ **execută**
6: mulţimea $h(v)$ primeşte valoarea $\delta(s,v)$ determinată de algoritmul Bellman-Ford
7: **pentru** fiecare arc $(u,v) \in E[G']$ **execută**
8: $\widehat{w}(u,v) \leftarrow w(u,v) + h(u) - h(v)$
9: **pentru** fiecare vârf $u \in V[G]$ **execută**
10: execută DIJKSTRA($G, \widehat{w}, u$) pentru a determina $\widehat{\delta}(u,v)$ pentru orice $v \in V[G]$
11: **pentru** fiecare vârf $v \in V[G]$ **execută**
12: $d_{uv} \leftarrow \widehat{\delta}(u,v) + h(v) - h(u)$
13: **returnează** D

Acest cod efectuează acţiunile specificate anterior. Linia 1 determină G'. Linia 2 execută algoritmul Bellman-Ford pe G' cu funcţia de cost w. Dacă G', deci şi G, conţine un ciclu de cost negativ, linia 3 raportează problema apărută. În liniile 4–11, se presupune că G' nu conţine cicluri de cost negativ. Liniile 4–6 atribuie lui $h(v)$ costul $\delta(u,v)$ al drumului minim determinat de algoritmul Belmann-Ford pentru toate vârfurile $v \in V'$. Liniile 7–8 determină noile costuri $\widehat{w}$. Pentru fiecare pereche de vârfuri $u, v \in V$, bucla **pentru** din liniile 9–12 determină costul $\widehat{\delta}(u,v)$, apelând algoritmul lui Dijkstra pentru fiecare vârf din V. Linia 12 stochează, în matrice,

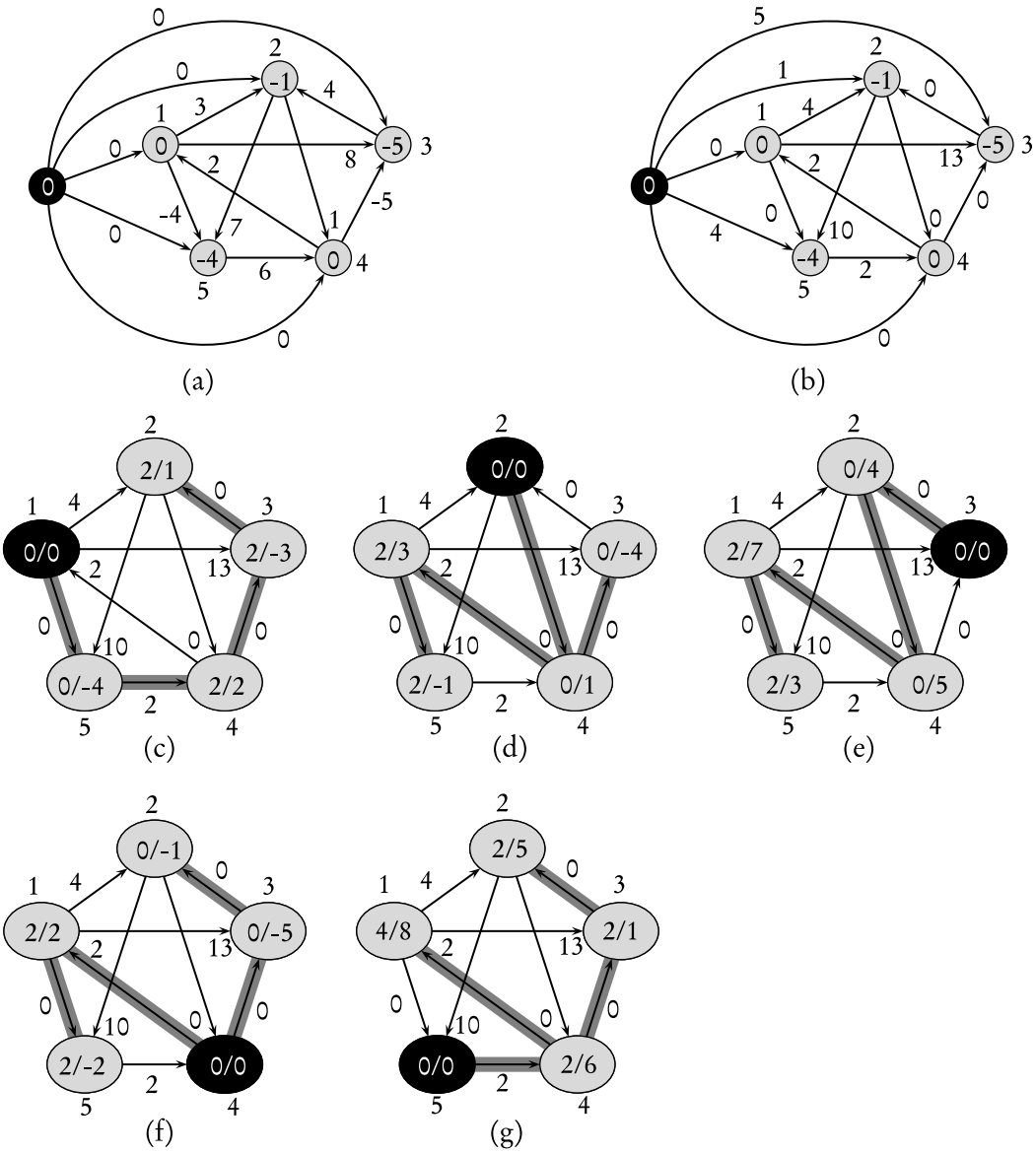


Figura 26.6 Algoritmul lui Johnson de determinare a drumurilor minime între toate perechile de vârfuri, executat pe graful din figura 26.1. **(a)** Graful G' cu funcţia originală w. Noul vârf s este negru. Pentru fiecare vârf v, avem $h(v) = \delta(s, v)$. **(b)** Costul fiecărui arc (u, v) este schimbat folosind funcţia de cost $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$. **(c)-(g)** Rezultatul executării algoritmului lui Dijkstra pe fiecare vârf al lui G folosind funcţia de cost $\widehat{w}$. În fiecare secţiune, vârful sursă u este negru. În interiorul fiecărui vârf v sunt scrise valorile $\widehat{\delta}(u, v)$ şi $\delta(u, v)$ separate printr-un slash. Valoarea $d_{uv} = \delta(u, v)$ este egală cu $\widehat{\delta}(u, v) + h(v) - h(u)$.

elementul d_{uv} corespunzător din matricea D, costul corect, al drumului minim calculat $\delta(u, v)$ folosind ecuaţia (26.10). În final, linia 13 returnează matricea D completată. În figura 26.6 este ilustrată execuţia algoritmului lui Johnson.

Se poate observa destul de uşor că timpul de execuţie al algoritmului lui Johnson este $O(V^2 \lg V + VE)$ dacă în algoritmul lui Dijkstra implementăm coada de prioritate cu ajutorul unui heap Fibonacci. Implementarea mai simplă folosind un heap binar duce la un timp de execuţie $O(VE \lg V)$, deci algoritmul este, asimptotic, mai rapid decât algoritmul Floyd-Warshall dacă graful este rar.

Exerciţii

26.3-1 Folosiţi algoritmul lui Johnson pentru a găsi drumurile minime dintre toate perechile de vârfuri în graful din figura 26.2. Specificaţi care sunt valorile lui h şi $\widehat{w}$ determinate de algoritm.

26.3-2 Care este scopul adăugării noului vârf s la V pentru a obţine V'?

26.3-3 Să presupunem că $w(u, v) \geq 0$ pentru toate arcele $(u, v) \in E$. Care este legătura dintre funcţiile de cost w şi $\widehat{w}$?

26.4. O modalitate generală pentru rezolvarea problemelor de drum în grafuri orientate

În această secţiune vom studia "semiinelele închise", o structură algebrică ce duce la o modalitate generală de a rezolva problemele de drum în grafuri orientate. Începem prin a defini semiinelele închise şi a discuta legătura lor cu determinarea drumurilor orientate. Vom arăta apoi câteva exemple de semiinele închise şi un algoritm "generic" pentru determinarea informaţiilor referitoare la drumurile între toate perechile. Atât algoritmul Floyd-Warshall cât şi algoritmul de determinare a închiderii tranzitive din secţiunea 26.2 sunt instanţe ale acestui algoritm generic.

Definiţia semiinelelor închise

Un ***semiinel închis*** este un tuplu $(S, \oplus, \odot, \overline{0}, \overline{1})$ unde S este mulţimea elementelor, $\oplus$ (***operatorul de însumare***) şi $\odot$ (***operatorul de extindere***) sunt operaţii binare peste S, şi $\overline{0}$ şi $\overline{1}$ sunt elemente din S care satisfac următoarele 8 proprietăţi:

1. $(S, \oplus, \overline{0})$ este un ***monoid***:
   - S este ***închisă*** faţă de $\oplus$: $a \oplus b \in S$ pentru orice $a, b \in S$.
   - $\oplus$ este ***asociativă***: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ pentru orice $a, b, c \in S$.
   - $\overline{0}$ este un ***element neutru*** pentru $\oplus$: $a \oplus \overline{0} = \overline{0} \oplus a = a$ pentru orice $a \in S$.

   De asemenea, $(S, \odot, \overline{1})$ este un monoid.
2. $\overline{0}$ este un ***anihilator***: $a \odot \overline{0} = \overline{0} \odot a = \overline{0}$, pentru orice $a \in S$.
3. $\oplus$ este ***comutativă***: $a \oplus b = b \oplus a$, pentru orice $a, b \in S$.

4. $\oplus$ este ***idempotentă***: $a \oplus a = a$ pentru orice $a \in S$.

5. $\odot$ este ***distributivă*** faţă de $\oplus$: $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ şi $(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$ pentru orice $a, b, c \in S$

6. Dacă $a_1, a_2, a_3, \ldots$ este un şir numărabil de elemente din S, atunci elementul $a_1 \oplus a_2 \oplus a_3 \oplus, \cdots$ este bine definit şi face parte din S.

7. Asociativitatea, comutativitatea şi idempotenţa se aplică pentru însumări infinite. (Deci orice însumare infinită poate fi rescrisă ca o însumare infinită în cadrul căreia fiecare termen apare numai o dată şi ordinea evaluării este arbitrară.)

8. $\odot$ este distributivă faţă de însumarea infinită: $a \odot (b_1 \oplus b_2 \oplus b_3 \oplus \cdots) = (a \odot b_1) \oplus (a \odot b_2) \oplus (a \odot b_3) \oplus \cdots$ şi $(a_1 \oplus a_2 \oplus a_3 \oplus \cdots) \odot b = (a_1 \odot b) \oplus (a_2 \odot b) \oplus (a_3 \odot b) \oplus \cdots$.

O determinare a drumurilor în grafuri orientate

Cu toate că proprietăţile semiinelelor închise par abstracte, ele pot fi puse în legătură cu drumurile în grafuri orientate. Să presupunem că se dă un graf orientat $G = (V, E)$ şi o ***funcţie de etichetare*** $\lambda : V \times V \to S$ care pune în corespondenţă toate perechile ordonate de vârfuri cu elemente din S. ***Eticheta arcului*** $(u, v) \in E$ este notată prin $\lambda(u, v)$. Deoarece λ este definită peste domeniul $V \times V$, eticheta $\lambda(u, v)$ are, de obicei, valoarea $\overline{0}$ dacă (u, v) nu este un arc de-al lui G (vom vedea imediat motivul acestei alegeri).

Folosim operatorul asociativ de extindere $\odot$ pentru a extinde noţiunea de etichete pentru drumuri. ***Eticheta drumului*** $p = \langle v_1, v_2, \ldots, v_k \rangle$ este

$$\lambda(p) = \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k).$$

Elementul neutru $\overline{1}$ faţă de $\odot$ va fi eticheta drumului vid.

Ca un exemplu pentru o aplicaţie a semiinelelor închise, vom folosi drumuri minime pentru grafuri cu arce de costuri nenegative. Codomeniul S este $\mathbb{R}^{\geq 0} \cup \{\infty\}$, unde $\mathbb{R}^{\geq 0}$ este mulţimea numerelor reale nenegative şi $\lambda(i, j) = w_{ij}$ pentru orice $i, j \in V$. Operatorul de extindere $\odot$ corespunde operatorului aritmetic $+$, deci eticheta drumului $p = \langle v_1, v_2, \ldots, v_k \rangle$ este

$$\lambda(p) = \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k) = w_{v_1,v_2} + w_{v_2,v_3} + \cdots + w_{v_{k-1},v_k} = w(p).$$

Nu este surprinzător faptul că rolul lui $\overline{1}$, elementul neutru pentru $\odot$, este luat de 0, elementul neutru pentru $+$. Notăm drumul vid prin ε; eticheta lui este $\lambda(\varepsilon) = w(\varepsilon) = 0 = \overline{1}$.

Deoarece operaţia de extindere $\odot$ este asociativă, putem defini eticheta concatenării a două drumuri într-un mod natural. Dându-se drumurile $p_1 = \langle v_1, v_2, \ldots, v_k \rangle$ şi $p_2 = \langle v_k, v_{k+1}, \ldots, v_l \rangle$ ***concatenarea*** lor este

$$p_1 \circ p_2 = \langle v_1, v_2, \ldots, v_k, v_{k+1}, \ldots v_l \rangle,$$

şi eticheta concatenării lor este

$$\begin{aligned}\lambda(p_1 \circ p_2) &= \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k) \odot \lambda(v_k, v_{k+1}) \odot \lambda(v_{k+1}, v_{k+2}) \odot \cdots \odot \lambda(v_{l-1}, v_l) \\ &= (\lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k)) \odot (\lambda(v_k, v_{k+1}) \odot \lambda(v_{k+1}, v_{k+2}) \odot \cdots \odot \lambda(v_{l-1}, v_l)) \\ &= \lambda(p_1) \odot \lambda(p_2).\end{aligned}$$

Operatorul de însumare $\oplus$, care este atât comutativ cât şi asociativ, este folosit pentru a ***însuma*** etichetele drumurilor. Aceasta înseamnă că valoarea $\lambda(p_1) \oplus \lambda(p_2)$ dă o însumare a etichetelor drumurilor p_1 şi p_2, însumare a cărei semantică este specifică aplicaţiei.

Scopul nostru este de a determina, pentru toate perechile de vârfuri $i, j \in V$, însumarea tuturor etichetelor drumurilor de la i la j:

$$l_{ij} = \bigoplus_{i \overset{p}{\rightsquigarrow} j} \lambda(p) \tag{26.11}$$

Avem nevoie de comutativitatea şi asociativitatea lui $\oplus$ pentru ca ordinea în care sunt însumate drumurile să nu conteze. Deoarece folosim anihilatorul $\overline{0}$ ca etichetă a unei perechi ordonate (u, v), care nu este arc în graf, orice drum care încearcă să treacă printr-un arc absent va avea eticheta $\overline{0}$.

Pentru drumurile minime, folosim min ca operator de însumare $\oplus$. Elementul neutru pentru min este ∞, şi ∞ este, într-adevăr, un anihilator pentru $+$: $a + \infty = \infty + a = \infty$ pentru orice $a \in \mathbb{R}^{\geq 0} \cup \{\infty\}$. Arcele absente vor avea costul ∞ şi, dacă oricare arc al unui drum va avea costul ∞, drumul va avea acelaşi cost.

Dorim ca operatorul de însumare $\oplus$ să fie idempotent, deoarece, din ecuaţia (26.11), observăm că $\oplus$ ar trebui să însumeze etichetele unei mulţimi de drumuri. Dacă p este un drum, atunci $\{p\} \cup \{p\} = \{p\}$; dacă însumăm drumul p cu el însuşi, eticheta rezultată ar trebui să fie eticheta lui p: $\lambda(p) \oplus \lambda(p) = \lambda(p)$.

Deoarece considerăm şi drumuri care s-ar putea să nu fie elementare, mulţimea etichetelor drumurilor dintr-un graf poate fi infinită, dar numărabilă. (Fiecare drum, elementar sau nu, conţine un număr finit de arce.) În concluzie, operatorul $\oplus$ ar trebui să poată fi aplicat pentru o mulţime infinită, dar numărabilă, de etichete ale drumurilor. Aceasta înseamnă că, dacă $a_1, a_2, a_3, \ldots$ este un şir numărabil de elemente din codomeniul S, atunci eticheta $a_1 \oplus a_2 \oplus a_3 \oplus \cdots$ ar trebui să fie bine definită şi să facă parte din S. Nu ar trebui să conteze ordinea în care însumăm etichetele drumurilor, deci asociativitatea şi comutativitatea ar trebui să fie respectate pentru însumări infinite. Mai mult, dacă însumăm aceeaşi etichetă de drum a de un număr infinit de ori, rezultatul ar trebui să fie a, deci idempotenţa ar trebui respectată şi pentru însumări infinite.

Întorcându-ne la exemplul drumurilor minime, vrem să ştim dacă min este aplicabilă la un şir infinit de valori din $\mathbb{R}^{\geq 0} \cup \{\infty\}$. De exemplu, este bine definită valoarea $\min_{k=1}^{\infty}\{1/k\}$? Răspunsul este afirmativ dacă ne gândim la operatorul min ca returnând cel mai mare minorant (infimul) al argumentelor sale, caz în care obţinem $\min_{k=1}^{\infty}\{1/k\} = 0$.

Pentru a determina etichete ale unor drumuri divergente avem nevoie de distributivitatea operatorului de extindere $\odot$ faţă de operatorul de însumare $\oplus$. După cum se arată în figura 26.7, presupunem că avem drumurile $u \overset{p_1}{\rightsquigarrow} v$, $v \overset{p_1}{\rightsquigarrow} x$ şi $v \overset{p_3}{\rightsquigarrow} y$. Cu ajutorul distributivităţii, putem însuma etichetele drumurilor $p_1 \circ p_2$ şi $p_1 \circ p_3$ calculând fie $(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(p_3))$ sau $\lambda(p_1) \odot (\lambda(p_2) \oplus \lambda(p_3))$.

Deoarece pot exista un număr infinit, dar numărabil de drumuri într-un graf, $\odot$ ar trebui să fie distributivă atât faţă de însumări infinite, cât şi faţă de însumări finite. În figura 26.8 pot fi văzute drumurile $u \overset{p_1}{\rightsquigarrow} v$ şi $v \overset{p_2}{\rightsquigarrow} x$ precum şi ciclul $v \overset{c}{\rightsquigarrow} v$. Trebuie să putem însuma drumurile $p_1 \circ p_2$, $p_1 \circ c \circ p_2$, $p_1 \circ c \circ c \circ p_2$, ... Distributivitatea lui $\odot$ faţă de însumări infinite, dar numărabile, ne dau

$$\begin{aligned}
&(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(c) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(c) \odot \lambda(c) \odot \lambda(p_2)) \oplus \cdots \\
&= \lambda(p_1) \odot (\lambda(p_2) \oplus (\lambda(c) \odot \lambda(p_2)) \oplus (\lambda(c) \odot \lambda(c) \odot \lambda(p_2)) \oplus \cdots) \\
&= \lambda(p_1) \odot (\overline{1} \oplus \lambda(c) \oplus (\lambda(c) \odot \lambda(c)) \oplus (\lambda(c) \odot \lambda(c) \odot \lambda(c)) \oplus \cdots) \odot \lambda(p_2).
\end{aligned}$$

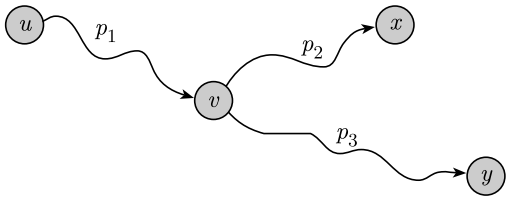


Figura 26.7 Folosirea distributivităţii lui $\odot$ faţă de $\oplus$. Pentru a însuma etichetele drumurilor $p_1 \circ p_2$ şi $p_1 \circ p_3$, putem calcula fie $(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(p_3))$, fie $\lambda(p_1) \odot (\lambda(p_2) \oplus \lambda(p_3))$.

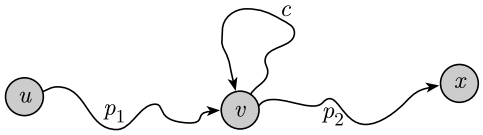


Figura 26.8 Distributivitatea lui $\odot$ faţă de însumări infinite, dar numărabile, ale lui $\oplus$. Datorită ciclului c, pot exista un număr infinit, dar numărabil, de drumuri de la vârful v la vârful x. Trebuie să putem însuma drumurile $p_1 \circ p_2$, $p_1 \circ c \circ p_2$, $p_1 \circ c \circ c \circ p_2$, ...

Trebuie să folosim o notaţie specială pentru eticheta unui ciclu care poate fi traversat de un anumit număr de ori. Să presupunem că avem un ciclu c cu eticheta $\lambda(c) = a$. Putem traversa c de zero ori, eticheta fiind, în acest caz, $\lambda(\varepsilon) = \overline{1}$. Dacă traversăm ciclul o dată, eticheta va fi $\lambda(c) = a$, dacă îl traversăm de două ori, eticheta va fi $\lambda(c) \odot \lambda(c) = a \odot a$, şi aşa mai departe. Eticheta pe care o obţinem însumând un număr infint de traversări ale ciclului c este ***închiderea*** lui a, definită prin

$$a^* = \overline{1} \oplus a \oplus (a \odot a) \oplus (a \odot a \odot a) \oplus (a \odot a \odot a \odot a) \oplus \cdots .$$

În concluzie, pentru figura 26.8, vrem să calculăm $\lambda(p_1) \odot (\lambda(c))^* \odot \lambda(p_2)$.

Pentru exemplul drumurilor minime, pentru orice număr real nenegativ din mulţimea $a \in \mathbb{R}^{\geq \infty} \cup \{\infty\}$,

$$a^* = \min_{k=0}^{\infty} \{ka\} = 0.$$

Interpretarea acestei proprietăţi este aceea că, deoarece toate ciclurile au costuri nenegative, nici un drum minim nu trebuie să traverseze un ciclu întreg.

Exemple de semiinele închise

Am văzut deja un exemplu de semiinel, $S_1 = (\mathbb{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$, pe care l-am folosit pentru drumuri minime cu arce de costuri nenegative. (După cum am amintit anterior, operatorul min returnează, de fapt, cel mai mare minorant al argumentelor sale. Am arătat, de asemenea, că $a^* = 0$ pentru orice $a \in \mathbb{R}^{\geq 0} \cup \{\infty\}$.

Am susţinut că, chiar dacă există arce de cost negativ, algoritmul Floyd-Warshall determină costurile drumurilor minime atâta timp cât nu există cicluri de cost negativ. Adăugând operatorul de închidere potrivit şi extinzând codomeniul etichetelor la $\mathbb{R} \cup \{-\infty, +\infty\}$ putem găsi un semiinel

închis care să trateze ciclurile de cost negativ. Folosind min pentru $\oplus$ şi + pentru $\odot$, cititorul poate verifica faptul că închiderea lui $a \in \mathbb{R} \cup \{-\infty, +\infty\}$ este

$$a^* = \begin{cases} 0 & \text{dacă } a \geq 0, \\ -\infty & \text{dacă } a < 0. \end{cases}$$

Al doilea caz ($a < 0$) modelează situaţia în care putem traversa un ciclu de cost negativ de un număr infinit de ori pentru a obţine costul $-\infty$ pentru orice drum care conţine cicluri. În concluzie, semiinelul închis care trebuie folosit pentru algoritmul Floyd-Warshall cu arce de cost negativ este $S_2 = (\mathbb{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$. (Vezi exerciţiul 26.4-3.)

Pentru închiderea tranzitivă, folosim semiinelul închis $S_3 = (\{0, 1\}, \vee, \wedge, 0, 1)$, unde $\lambda(i, j) = 1$ dacă $(i, j) \in E$, şi $\lambda(i, j) = 0$ în caz contrar. Aici avem $0^* = 1^* = 1$.

Un algoritm de programare dinamică pentru determinarea etichetelor drumurilor orientate

Să presupunem că se dă un graf orientat $G = (V, E)$ cu funcţia de etichetare $\lambda : V \times V \to S$. Vârfurile sunt numerotate de la 1 la n. Pentru fiecare pereche de vârfuri $i, j \in V$, vrem să calculăm ecuaţia (26.11)

$$l_{ij} = \bigoplus_{i \overset{p}{\rightsquigarrow} j} \lambda(p),$$

care este rezultatul însumării tuturor drumurilor de la i la j, folosind operatorul de însumare $\oplus$. Pentru drumurile minime, vrem să calculăm

$$l_{ij} = \delta(i, j) = \min_{i \overset{p}{\rightsquigarrow} j} \{w(p)\}.$$

Există un algoritm de programare dinamică pentru rezolvarea acestei probleme şi acesta seamănă foarte mult cu algoritmul Floyd-Warshall şi cu algoritmul de determinare a închiderii tranzitive. Fie $Q_{ij}^{(k)}$ mulţimea drumurilor de la vârful i la vârful j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$. Definim

$$l_{ij}^{(k)} = \bigoplus_{p \in Q_{ij}^{(k)}} \lambda(p).$$

Observaţi analogia cu definiţia lui $d_{ij}^{(k)}$ din algoritmul Floyd-Warshall şi cu $t_{ij}^{(k)}$ din algoritmul de determinare a închiderii tranzitive. Putem defini recursiv $l_{ij}^{(k)}$ prin

$$l_{ij}^{(k)} = l_{ij}^{(k-1)} \oplus \left(l_{ik}^{(k-1)} \odot (l_{kk}^{(k-1)})^* \odot l_{kj}^{(k-1)} \right). \tag{26.12}$$

Recurenţa (26.12) seamănă cu recurenţele (26.5) şi (26.8), dar conţine un nou factor: $(l_{kk}^{(k-1)})^*$. Acest factor reprezintă însumarea tuturor ciclurilor care trec prin vârful k şi au toate celelalte vârfuri în mulţimea $\{1, 2, \ldots, k-1\}$. (Când presupunem, în algoritmul Floyd-Warshall, că nu există cicluri de cost negativ, $(c_{kk}^{(k-1)})^*$ este 0, corespunzător lui $\overline{1}$, costul unui ciclu vid.) În cadrul algoritmului de determinare a închiderii tranzitive, drumul vid de la k la k ne dă $(t_{kk}^{(k-1)})^* =$

$1 = \overline{1}$. Deci, pentru fiecare dintre aceşti doi algoritmi, putem ignora factorul $(l_{kk}^{(k-1)})^*$ deoarece reprezintă elementul neutru pentru $\odot$. Baza definiţiei recursive este

$$l_{ij}^{(0)} = \begin{cases} \lambda(i,j) & \text{dacă } i \neq j, \\ \overline{1} \oplus \lambda(i,j) & \text{dacă } i = j, \end{cases}$$

pe care o putem vedea după cum urmează. Eticheta unui drum care conţine un singur arc $\langle i, j \rangle$ este, simplu, $\lambda(i,j)$ (care este egală cu $\overline{0}$ dacă (i,j) nu este o muchie din E). Dacă $i = j$, atunci $\overline{1}$ este eticheta drumului vid de la i la i.

Algoritmul de programare dinamică determină valorile $l_{ij}^{(k)}$ în ordinea crescătoare a lui k. El returnează matricea $L^{(n)} = \left(l_{ij}^{(n)}\right)$.

CALCULEAZĂ-ÎNSUMĂRI(λ, V)

```
1:  n ← |V|
2:  pentru i ← 1, n execută
3:    pentru j ← 1, n execută
4:      dacă i = j atunci
5:        l_ij^(0) ← 1̄ ⊕ λ(i, j)
6:      altfel
7:        l_ij^(0) ← λ(i, j)
8:  pentru k ← 1, n execută
9:    pentru i ← 1, n execută
10:     pentru j ← 1, n execută
11:       l_ij^(k) ← l_ij^(k−1) ⊕ (l_ik^(k−1) ⊙ (l_kk^(k−1))* ⊙ l_kj^(k−1))
12: returnează L^(n)
```

Timpul de execuţie al acestui algoritm depinde de timpul necesar calculării lui $\odot, \oplus$ şi *. Dacă $T_\odot$, $T_\oplus$ şi T_* reprezintă aceşti timpi, atunci timpul de execuţie al algoritmului CALCULEAZĂ-ÎNSUMĂRI este $\Theta(n^3(T_\odot + T_\oplus + T_*))$ ceea ce reprezintă $\Theta(n^3)$ dacă toate aceste trei operaţii necesită un timp $O(1)$.

Exerciţii

26.4-1 Verificaţi că $S_1 = (\mathbb{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$ şi $S_3 = (\{0,1\}, \vee, \wedge, 0, 1)$ sunt semiinele închise.

26.4-2 Verificaţi că $S_2 = (\mathbb{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$ este un semiinel închis. Care este valoarea lui $a + (-\infty)$ pentru $a \in \mathbb{R}$? Dar a lui $(-\infty) + (+\infty)$?

26.4-3 Rescrieţi procedura CALCULEAZĂ-ÎNSUMĂRI pentru a folosi semiinelul închis S_2 astfel încât să implementeze algoritmul Floyd-Warshall. Care ar trebui să fie valoarea lui $-\infty + \infty$?

26.4-4 Este tuplul $S_4 = (\mathbb{R}, +, \cdot, 0, 1)$ un semiinel închis?

26.4-5 Putem folosi un semiinel închis arbitrar pentru algoritmul lui Dijkstra? Dar pentru algoritmul Bellman-Ford? Dar pentru procedura DRUMURI-MINIME-ÎNTRE-TOATE-PERECHILE-MAI-RAPID?

26.4-6 O firmă de transport doreşte să trimită din Castroville la Boston un camion încărcat până la refuz cu anumite articole. Fiecare şosea din Statele Unite are o anumită limită pentru camioanele care pot folosi acea şosea. Modelaţi această problemă pe un graf orientat $G = (V, E)$ şi un semiinel închis adecvat şi daţi un algoritm eficient pentru rezolvarea ei.

Probleme

26-1 Închiderea tranzitivă a unui graf dinamic

Să presupunem că dorim să menţinem închiderea tranzitivă a unui graf orientat $G = (V, E)$ în timp ce inserăm noi arce în E. Mai precis, după ce a fost inserat un arc, vrem să actualizăm închiderea tranzitivă a arcelor inserate până în acel moment. Să presupunem că, iniţial, graful G nu are nici un arc şi că închiderea tranzitivă este reprezentată de o matrice booleană.

a. Arătaţi modul în care închiderea tranzitivă $G^* = (V, E^*)$ a unui graf $G = (V, E)$ poate fi actualizată în timp $O(V^2)$ când în G se adaugă un arc nou.

b. Daţi un exemplu de graf G şi de o muchie e, astfel încât să fie nevoie de un timp $\Omega(V^2)$ pentru actualizarea închiderii tranzitive după inserarea lui e în G.

c. Descrieţi un algoritm eficient pentru actualizarea închiderii tranzitive pe măsură ce noi arce sunt adăugate grafului. Pentru orice secvenţă de n inserări, algoritmul ar trebui să se execute într-un timp total $\sum_{i=1}^{n} t_i = O(V^3)$, unde t_i este timpul necesar pentru actualizarea închiderii tranzitive când al i-lea arc este inserat. Demonstraţi că algoritmul atinge această limită de timp.

26-2 Drumuri minime în grafuri ϵ-dense

Un graf $G = (V, E)$ este ***ϵ-dens*** dacă $|E| = \Theta(V^{1+\epsilon})$ pentru o anumită constantă ϵ din domeniul $0 < \epsilon \leq 1$. Folosind un heap d-ar (vezi problema 7-2) în algoritmii de drumuri minime pe grafuri ϵ-dense, putem obţine timpi de execuţie la fel de buni ca în cazul algoritmilor bazaţi pe heap-uri Fibonacci fără să folosim o structură de date atât de complicată.

a. Care este timpul de execuţie asimptotic pentru INSEREAZĂ, EXTRAGE-MIN şi DESCREŞTE-CHEIE ca funcţie de d şi de numărul n al elementelor din heap-ul d-ar? Care sunt aceşti timpi de execuţie dacă alegem $d = \Theta(n^\alpha)$ pentru o anumită constantă $0 < \alpha \leq 1$? Comparaţi costul amortizat al acestor operaţii folosind un heap Fibonacci cu timpii de execuţie determinaţi anterior.

b. Arătaţi modul în care pot fi determinate, în timp $O(E)$, drumuri minime de la o singură sursă pe un graf orientat $G = (V, E)$ ϵ-dens fără arce de cost negativ. (*Indicaţie:* Alegeţi d ca fiind o funcţie de ϵ.)

c. Arătaţi modul în care poate fi rezolvată, în timp $O(VE)$, problema drumurilor minime pe un graf orientat $G = (V, E)$ ϵ-dens fără arce de cost negativ.

d. Arătaţi modul în care poate fi rezolvată, în timp $O(VE)$, problema drumurilor minime între toate perechile de vârfuri pe un graf orientat $G = (V, E)$ ϵ- dens care poate avea arce de cost negativ, dar nu are nici un ciclu de cost negativ.

26-3 Arbore parţial minim ca semiinel închis
Fie $G = (V, E)$ un graf neorientat conex cu funcţia de cost $w : E \to \mathbb{R}$. Fie $V = \{1, 2, \ldots, n\}$ mulţimea vârfurilor, unde $n = |V|$ şi să presupunem că toate costurile $w(i, j)$ ale muchiilor sunt unice. Fie T unicul arbore parţial minim (vezi exerciţiul 24.1-6) al grafului G. În cadrul acestei probleme, vom determina T folosind un semiinel închis după cum au sugerat B. M. Maggs şi S. A. Plotkin. Determinăm, mai întâi, pentru fiecare pereche de vârfuri $i, j \in V$, costul ***minimax***

$$m_{ij} = \min_{i \overset{p}{\rightsquigarrow} j} \max_{\text{muchii } e \text{ din } p} w(e).$$

a. Justificaţi, pe scurt, afirmaţia că $S = (\mathbb{R} \cup \{-\infty, \infty\}, \min, \max, \infty, -\infty)$ este un semiinel închis.

Deoarece S este un semiinel închis, putem folosi procedura CALCULEAZĂ-ÎNSUMĂRI pentru a determina costurile minimax m_{ij} în graful G. Fie $m_{ij}^{(k)}$ costul minimax peste toate drumurile de la i la j cu toate vârfurile intermediare în mulţimea $\{1, 2, \ldots, k\}$.

b. Daţi o recurenţă pentru $m_{ij}^{(k)}$, unde $k \geq 0$.

c. Fie $T_m = \{(i, j) \in E : w(i, j) = m_{ij}\}$. Demonstraţi că muchiile din T_m formează un arbore parţial al lui G.

d. Arătaţi că $T_m = T$. (*Indicaţie:* Luaţi în considerare efectul adăugării la T a unei muchii (i, j) şi al eliminării unei muchii de pe un alt drum de la i la j. De asemenea, luaţi în considerare efectul eliminării unei muchii (i, j) din T şi înlocuirii ei cu o altă muchie.)

Note bibliografice

Lawler [132] tratează foarte bine problema drumurilor minime între toate perechile de vârfuri, cu toate că nu analizează soluţiile pentru grafuri rare. El atribuie folclorului algoritmul de înmulţire a matricelor. Algoritmul Floyd-Warshall este datorat lui Floyd [68] care s-a bazat pe o teoremă a lui Warshall [198] care descrie modul în care poate fi determinată închiderea tranzitivă pe matrice booleene. Structura algebrică a semiinelului închis apare în Aho, Hopcroft şi Ullman [4]. Algoritmul lui Johnson este preluat din [114].

27 Flux maxim

Aşa cum se poate modela o hartă rutieră printr-un graf orientat pentru a afla cel mai scurt drum de la un punct la un altul, graful orientat poate fi considerat ca o "reţea de transport" care dă răspunsuri la întrebări referitoare la fluxuri materiale. Să ne imaginăm o situaţie în care un material este transportat într-un sistem de la sursă, unde este produs, la destinaţie, unde este consumat. La sursă se produce materialul într-un ritm constant, iar la destinaţia se consumă în acelaşi ritm. Intuitiv, "fluxul" materialului, în orice punct al sistemului, este ritmul în care materialul se deplasează. Reţelele de transport pot modela scurgerea lichidului în sisteme cu ţevi, deplasarea pieselor pe benzi rulante, scurgerea curentului prin reţele electrice, deplasare informaţiilor prin reţele de comunicaţii, şi multe altele.

Fiecare arc în reţeaua de transport poate fi considerat drept conductă pentru material. Fiecare conductă are o capacitate dată, care este de fapt ritmul maxim cu care lichidul se poate deplasa în conductă. De exemplu, printr-o ţeavă pot curge cel mult 2000 litri de apă pe oră, sau pe un fir conductor un curent electric de maximum 20 amperi. Vârfurile (nodurile) sunt joncţiunile conductelor şi în afara vârfului sursă şi destinaţie, materialul nu se poate acumula în nici un vârf. Altfel spus, cantitatea de material care intră într-un vârf trebuie să fie egală cu cea care iese din vârf. Această proprietate se numeşte "conservarea fluxului" şi este identică cu legea lui Kirchoff în cazul curentului electric.

Problema fluxului maxim este cea mai simplă problemă legată de reţele de transport, care cere să se determine cantitatea maximă de material care poate fi transportată de la sursă la destinaţie ţinând cont de restricţiile de capacitate. După cum vom vedea în acest capitol, problema se poate rezolva prin algoritmi eficienţi. Mai mult, tehnicile de bază ale acestor algoritmi pot fi adaptate la rezolvarea altor tipuri de probleme în reţele de transport.

Acest capitol prezintă două metode generale pentru rezolvarea problemei fluxului maxim. Secţiunea 27.1 prezintă formalizarea noţiuniilor de reţea de transport şi flux în reţea, definind formal şi problema fluxului maxim. Secţiunea 27.2 descrie metoda clasică a lui Ford-Fulkerson pentru găsirea fluxului maxim. O aplicaţie a acestei metode este cea de găsire a cuplajului maxim într-un graf bipartit neorientat care se prezintă în secţiunea 27.3. Secţiunea 27.4 se ocupă de metoda preflux, care este cea mai rapidă metodă referitoare la probleme de flux. În secţiunea 27.5 se tratează o implementare particulară a algoritmului preflux, care are complexitatea $O(V^3)$. Cu toate că acesta nu este cel mai rapid algoritm cunoscut, el ilustrează bine unele tehnici folosite în algoritmi care asimptotic sunt cei mai rapizi şi are o comportare rezonabilă în practică.

27.1. Reţele de transport

În această secţiune vom da o definiţie cu ajutorul grafurilor a reţelelor de transport, vom discuta proprietăţile lor şi vom defini problema fluxului maxim. Vom introduce şi unele notaţii utile.

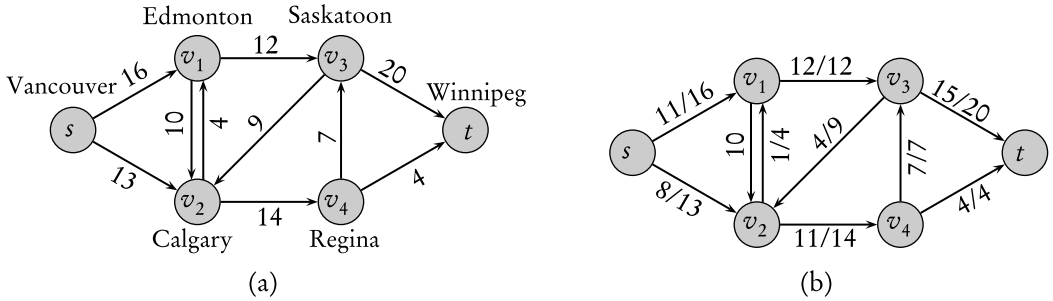


Figura 27.1 (**a**) O reţea $G = (V, E)$ pentru problema de transport al firmei Lucky Puck Company. Fabrica de pucuri din Vancouver este sursa s, iar depozitul din Winnipeg destinaţia t. Pucurile sunt transportate prin localităţile intermediare, dar numai $c(u, v)$ lăzi pe zi pot fi transportate din localitatea u în localiatatea v. Fiecare arc este marcat cu valoarea capacităţii respective. (**b**) Un flux f în G de valoare $|f| = 19$. Numai fluxurile de reţea pozitive sunt marcate pe arce. Dacă $f(u, v) > 0$ atunci arcul (u, v) se marchează cu $f(u, v)/c(u, v)$. (Aici notaţia / se foloseşte pentru a separa cele două valori – fluxul şi capacitatea – , şi nicidecum nu înseamnă împărţire.) Dacă $f(u, v) \leq 0$ atunci arcul (u, v) este marcat numai cu capacitate.

Reţele de transport şi fluxuri

O ***reţea de transport*** $G = (V, E)$ este un graf orientat în care fiecărui arc $(u, v) \in E$ îi este asociată o ***capacitate*** nenegativă $c(u, v) \geq 0$. Dacă $(u, v) \notin E$ vom considera că $c(u, v) = 0$. Vom distinge două vârfuri în reţea: un vârf ***sursă*** s şi un vârf ***destinaţie*** t. Vom presupune că fiecare vârf se găseşte pe cel puţin un drum de la sursă la destinaţie. Adică, pentru orice vârf $v \in V$ există un drum $s \rightsquigarrow v \rightsquigarrow t$. Graful este deci conex şi $|E| \geq |V| - 1$. Figura 27.1 prezintă un exemplu de reţea de transport.

Acum suntempregătiţi să dăm o definiţie mai formală a fluxului. Fie $G = (V, E)$ o reţea de transport cu o funcţie de capacitate c. Fie s vârful sursă şi t vârful destinaţie. ***Fluxul*** în G este o funcţie $f : V \times V \to \mathbb{R}$ cu valori reale care satisface următoarele trei condiţii:
Restricţie de capacitate: Pentru orice $u, v \in V$ avem $f(u, v) \leq c(u, v)$.
Antisimetrie: Pentru orice $u, v \in V$ avem $f(u, v) = -f(v, u)$.
Conservarea fluxului: Pentru orice $u \in V - \{s, t\}$ avem

$$\sum_{v \in V} f(u, v) = 0.$$

Cantitatea $f(u, v)$ care poate fi pozitivă sau negativă se numeşte ***fluxul net*** de la vârful u la vârful v, sau pur şi simplu ***fluxul*** pe arcul (u, v). ***Valoarea*** fluxului f se defineşte ca:

$$|f| = \sum_{v \in V} f(s, v), \tag{27.1}$$

adică fluxul total care pleacă din vârful sursă. (Aici prin semnul $|\cdot|$ se notează valoarea fluxului şi nu valoarea absolută sau cardinalul unei mulţimi.) Fiind dată o reţea de transport G cu sursa s şi destinaţia t, ***problema fluxului maxim*** cere găsirea unui flux de valoare maximă de la s la t.

Înainte de a vedea un exemplu de problemă de reţea de transport, să studiem pe scurt cele trei proprietăţi ale fluxului. Restricţia de capacitate impune pur şi simplu ca fluxul de la un vârf la altul să nu depăşească valoarea capacităţii date. Antisimetria impune ca fluxul de la un vârf u la un vârf v să fie egal cu opusul fluxului de la vârful v la u. Astfel, fluxul de la un vârf la el însuşi este egal cu 0, deoarece pentru orice $u \in V$ avem $f(u,u) = -f(u,u)$, care implică $f(u,u) = 0$. Proprietatea de conservare a fluxului cere ca fluxul total ce pleacă dintr-un vârf diferit de sursă şi destinaţie să fie egal cu 0. Plecând de la antisimetrie, putem rescrie proprietatea de conservare a fluxului ca

$$\sum_{u \in V} f(u,v)$$

pentru orice $v \in V - \{s,t\}$. Adică, în fiecare vârf fluxul total este egal cu 0.

Să observăm că fluxul între vârfurile u şi v care nu sunt legate prin nici un arc nu poate fi decât zero. Dacă $(u,v) \notin E$ şi $(v,u) \notin E$ atunci $c(u,v) = c(v,u) = 0$. Şi atunci, din cauza restricţiei de capacitate avem $f(u,v) \leq 0$ şi $f(v,u) \leq 0$. Dar, deoarece $f(u,v) = -f(v,u)$ din cauza antisimetriei, rezultă că $f(u,v) = f(v,u) = 0$. Deci, existenţa fluxului nenul între vârfurile u şi v implică $(u,v) \in E$ sau $(v,u) \in E$ sau ambele.

Ultima noastră observaţie, privind proprietăţile fluxului, se referă la fluxuri pozitive. ***Fluxul pozitiv*** care intră în vârful v se defineşte ca

$$\sum_{\substack{u \in V \\ f(u,v) > 0}} f(u,v). \tag{27.2}$$

Se poate defini în mod asemănător şi fluxul pozitiv care pleacă dintr-un vârf. O interpretare a conservării fluxului poate fi următoarea: fluxul pozitiv care intră într-un vârf trebuie să fie egal cu fluxul pozitiv care iese din vârf.

Un exemplu de reţea de transport

O reţea de transport poate modela problema de transport din figura 27.1. Firma Lucky Puck Company are o fabrică (sursa s) în oraşul Vancouver care fabrică pucuri pentru hockei pe gheaţă, şi are un depozit (destinaţia t) în oraşul Winnipeg unde le stochează. Firma închiriază spaţiu în camioane de la o altă firmă pentru a transporta pucurile de la fabrică la depozit. Pentru că spaţiul este limitat, firma poate transporta zilnic între oricare două localităţi u şi v numai $c(u,v)$ lăzi de pucuri (figura 27.1(a)). Firma Lucky Puck Company nu poate controla nici rutele de transport şi nici capacităţile, deci nu poate modifica reţeaua dată în figura 27.1(a). Scopul firmei este de a determina numărul maxim de lăzi p care pot fi transportate zilnic la depozit pentru a fabrică exact cantitatea respectivă, deoarece la fabrică nu au spaţiu pentru depozitare.

Ritmul cu care se deplasează pucurile de-alungul rutelor este un flux. Pucurile sunt transmise din fabrică cu un ritm de p lăzi pe zi, şi p lăzi trebuie să ajungă la depozit în fiecare zi. Firma Lucky Puck Company nu se ocupă de timpul de deplasare a pucurilor prin reţea, ei asigură doar ca p lăzi să părăsească zilnic fabrica şi p lăzi să ajungă la depozit. Restricţia de capacitate impune ca fluxul $f(u,v)$ de la oraşul u la oraşul v să nu depăşească $c(u,v)$ lăzi pe zi. Ritmul cu care ajung lăzile într-un oraş intermediar trebuie să fie identic cu ritmul cu care ele părăsesc oraşul respectiv, altfel s-ar îngrămădi. Deci, conservarea fluxului este respectată. Astfel, fluxul maxim în reţea determină numărul maxim p de lăzi transportate zilnic.

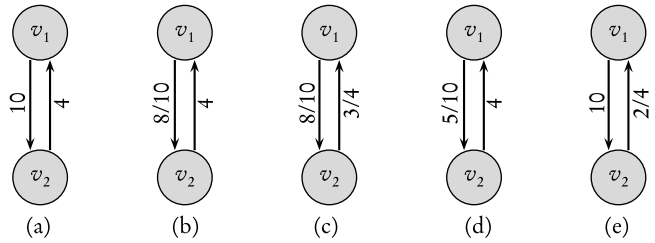


Figura 27.2 Anulare. (**a**) Vârfurile v_1 şi v_2 cu $c(v_1, v_2) = 10$ şi $c(v_2, v_1) = 4$. (**b**) Indicarea faptului că se transportă 8 lăzi pe zi de la v_1 la v_2. (**c**) Se mai transportă 3 lăzi pe zi de la v_2 la v_1. (**d**) Prin anularea fluxului care merge în direcţie opusă se poate reprezenta situaţia din (c) prin numai fluxuri pozitive într-o singură direcţie. (**e**) Alte 7 lăzi care sunt transportate de la v_2 la v_1.

Figura 27.1(b) ilustrează un flux posibil în reţeaua care modelează transportul lăzilor. Pentru oricare două vârfuri u şi v, fluxul $f(u, v)$ corespunde numărul lăzilor care pot fi transportate într-o zi de la u la v. Dacă $f(u, v)$ este 0 sau negativ, atunci nu există transport de la u la v. În figura 27.1(b) sunt marcate cu două valori (flux/capacitate) doar arcele cu flux pozitiv.

Relaţia dintre flux şi transport se poate înţelege mai bine dacă ne concentrăm atenţia asupra transportului între două vârfuri. Figura 27.2(a) ne arată subgraful indus de vârfurile v_1 şi v_2 din reţeaua de transport din figura 27.1. Dacă firma Lucky Puck transportă 8 lăzi pe zi de la v_1 la v_2 rezultatul este arătat în figura 27.2(b): fluxul de la v_1 la v_2 este de 8 lăzi pe zi. Din cauza antisimetriei se poate spune că fluxul în direcţie opusă, adică de la v_2 la v_1, este -8 lăzi pe zi, chiar dacă nu se transportă nimic de la v_2 la v_1. În general, fluxul de la v_1 la v_2 este exprimat prin numărul de lăzi transportate zilnic de la v_1 la v_2 din care se scade numărul de lăzi transportate de la v_2 la v_1. Convenţia noastră este de a reprezenta numai fluxuri pozitive, deoarece ele indică un transport real, astfel în figură apare numai valoarea 8, fără valoarea corespunzătoare -8.

Să mai adăugăm un transport de 3 lăzi pe zi de la v_2 la v_1. O reprezentare naturală a acestei situaţii se găseşte în figura 27.2(c). Apare acum o situaţie în care avem transport în ambele direcţii. Se transportă zilnic 8 lăzi de la v_1 la v_2 şi 3 lăzi de la v_2 la v_1. Fluxul de la v_1 la v_2 este acum de $8 - 3 = 5$ lăzi pe zi, iar fluxul de la v_2 la v_1 este de $3 - 8 = -5$ lăzi.

Situaţia, datorită rezultatului, este echivalentă cu cea arătată în figura 27.2(d), unde sunt transportate 5 lăzi de la v_1 la v_2, şi nimic de la v_2 la v_1. De fapt, 3 lăzi pe zi de la v_2 la v_1 sunt ***anulate*** de 3 lăzi din 8 transportate de la v_1 la v_2. În ambele situaţii, fluxul de la v_1 la v_2 este de 5 lăzi pe zi, dar în figura (d) sunt arătate numai transporturile într-o singură direcţie.

În general, anulările ne permit să reprezentăm transportul între două vârfuri printr-un flux pozitiv de-alungul a cel mult unuia dintre cele două arce. Dacă există flux zero sau negativ de la un vârf la altul, nu se face nici un transport în direcţia respectivă. Astfel, dacă există transport în ambele direcţii între două vârfuri, prin procesul de anulare situaţia poate fi transformată într-una echivalentă în care există transport numai într-o singură direcţie: direcţia fluxului pozitiv. Restricţia de capacitate nu este încălcată în această situaţie, deoarece fluxul între cele două vârfuri rămână aceeaşi.

Continuând cu exemplul nostru, să determinăm efectul transportării a încă 7 lăzi pe zi de la v_2 la v_1. Figura 27.2(e) ne arată rezultatul, ţinând cont de convenţia de reprezentare numai a fluxului pozitiv. Fluxul de la v_1 la v_2 devine $5 - 7 = -2$, iar fluxul de la v_2 la v_1 devine $7 - 5 = 2$. Deoarece fluxul de la v_2 la v_1 este pozitiv, el reprezintă un transport de 2 lăzi pe zi de la v_2 la

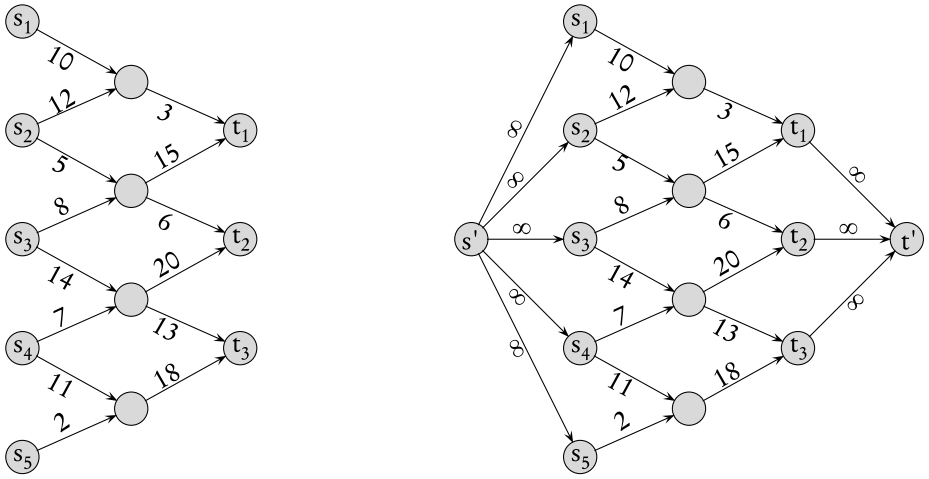


Figura 27.3 Transformarea unei probleme de flux maxim cu mai multe surse şi destinaţii într-o problemă de flux maxim cu o singură sursă şi destinaţie. (**a**) O reţea de transport cu cinci surse $S = \{s_1, s_2, s_3, s_4, s_5\}$ şi trei destinaţii $T = \{t_1, t_2, t_3\}$. (**b**) O reţea de transport echivalentă care are o singură sursă şi o singură destinaţie. Adăugăm o sursă nouă s' care este legat cu fiecare sursă originală printr-un arc de capacitate ∞. De asemena, adăugăm o nouă destinaţie t' şi arce având capacitatea ∞, de la fiecare destinaţie iniţială la cea nouă.

v_1. Fluxul de la v_1 la v_2 este -2 lăzi pe zi, şi, deoarece fluxul nu este pozitiv, nu se realizează nici un transport în această direcţie. Sau, putem considera că 5 lăzi din cele 7 lăzi adiţionale pe zi de la v_2 la v_1 sunt anulate de cele 5 lăzi de la v_1 la v_2, ceea ce ne conduce la un transport de 2 lăzi pe zi de la v_2 la v_1.

Reţele cu mai multe surse şi destinaţii

O problemă de flux maxim poate avea mai multe surse şi mai multe destinaţii în loc de exact una din fiecare. De exemplu, firma Lucky Puck poate avea o mulţime de m fabrici $\{s_1, s_2, \ldots, s_m\}$ şi o mulţime de n depozite $\{t_1, t_2, \ldots, t_n\}$ cum se arată în figura 27.3. Din fericire, această problemă nu este mai grea decât cea iniţială.

Problema găsirii unui flux maxim într-o reţea de transport cu mai multe surse şi destinaţii o vom putea reduce la problema de flux maxim într-o reţea cu o singură sursă şi destinaţie. Figura 27.3(b) ilustrează transformarea reţelei de transport cu mai multe surse şi destinaţii din figura 27.3(a) într-una cu o singură sursă şi destinaţie. Adăugăm o sursă nouă (***supersursă***) s şi arcele orientate (s, s_i) cu capacitatea $c(s, s_i) = \infty$ pentru fiecare $i = 1, 2, \ldots, m$. De asemena, se adaugă o nouă destinaţie (***superdestinaţie***) t şi arcele orientate (t_i, t) cu capacitatea $c(t_i, t) = \infty$ pentru fiecare $i = 1, 2, \ldots, n$. Intuitiv, oricărui flux în reţeaua din (a) îi corespunde un flux în reţeaua din (b) şi viceversa. Sursa s produce cantitatea cerută de sursele s_i, iar destinaţia t consumă cantitatea cerută de destinaţiile t_i. Exerciţiul 27.1-3 cere demonstrarea formală a echivalenţei de mai sus.

Câteva egalităţi referitoare la flux

Ne vom ocupa în continuare de unele funcţii (asemănătoare funcţiei f) care au ca argumente două vârfuri din reţeaua de transport. Vom folosi, în acest capitol, o ***notaţie de însumare implicită*** în care oricare din argumentele funcţiei (chiar şi ambele) pot fi *mulţimi* de vârfuri, cu interpretarea că valoarea respectivă este suma tuturor valorilor funcţiei pe toate perechile de elemente posibile din cele două mulţimi argumente. De exemplu, dacă X şi Y sunt mulţimi de vârfuri, atunci

$$f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y).$$

Ca un alt exemplu, se poate exprima cu ajutorul acestei notaţii restricţia de conservare a fluxului în felul următor: $f(u,V) = 0$ pentru orice $u \in V - \{s,t\}$. Pentru o scriere mai simplă vom omite acoladele când mulţimea are un singur element (cum am şi făcut-o în exemplul anterior). În formula $f(s, V - s) = f(s,V)$, de exemplu, termenul $V - s$ înseamnă $V - \{s\}$.

Această notaţie simplifică formulele în care intervin fluxuri. Lema următoare, a cărei demonstraţie este lăsată ca exerciţiu (vezi exerciţiul 27.1-4), cuprinde majoritatea identităţilor care intervin în probleme de flux şi folosesc notaţia de însumare implicită.

Lema 27.1 Fie $G = (V,E)$ o reţea de transport şi f un flux în G. Atunci, pentru $X \subseteq V$, avem $f(X,X) = 0$.
Pentru $X, Y \subseteq V$, avem $f(X,Y) = -f(Y,X)$.
Pentru $X, Y, Z \subseteq V$, cu $X \cap Y = \emptyset$, avem $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$ şi $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$. ∎

Ca exemplu de folosire a notaţiei de însumare implicită să demonstrăm că valoarea fluxului este egală cu fluxul total ce ajunge în vârful destinaţie, adică

$$|f| = f(V,t). \tag{27.3}$$

Se vede intuitiv că această formulă este adevărată. Toate vârfurile diferite de sursă şi de destinaţie au un flux 0 prin conservarea fluxului şi astfel vârful destinaţie este singurul care poate avea un flux diferit de 0 pentru a echivala fluxul diferit de 0 al sursei. O demonstraţie formală este următoarea:

$$\begin{aligned}
|f| &= f(s,V) && \text{(pe baza definiţiei)} \\
&= f(V,V) - f(V-s,V) && \text{(pe baza lemei 27.1)} \\
&= f(V,V-s) && \text{(pe baza lemei 27.1)} \\
&= f(V,t) + f(V,V-s-t) && \text{(pe baza lemei 27.1)} \\
&= f(V,t) && \text{(pe baza legii de conservare a fluxului)}
\end{aligned}$$

Acest rezultat va fi genrealizat mai târziu (lema 27.5).

Exerciţii

27.1-1 Date fiind vârfurile u şi v într-o reţea de transport, având capacităţile $c(u,v) = 5$ şi $c(v,u) = 8$, să presupunem că avem un flux egal cu 3 unităţi de la u la v şi un flux egal cu 4 unităţi de la v la u. Care este fluxul de la u la v? Desenaţi situaţia în stilul figurii 27.2.

27.1-2 Verificaţi toate cele trei proprietăţi pentru fluxul f din figura 27.1(b).

27.1-3 Extindeţi proprietăţile fluxului pentru cazul reţelei cu mai multe surse şi destinaţii. Arătaţi că orice flux într-o reţea de transport cu mai multe surse şi destinaţii corespunde unui flux cu aceeaşi valoare într-o reţea cu o singură sursă şi o singură destinaţie, prin adăugarea unei supersurse şi a unei superdestinaţii.

27.1-4 Demonstraţi lema 27.1

27.1-5 Fie reţeaua de transport $G = (V, E)$ din figura 27.1(b). Găsiţi o pereche de submulţimi $X, Y \subseteq V$ pentru care $f(X, Y) = -f(V - X, Y)$. Apoi, găsiţi o pereche de submulţimi $X, Y \subseteq V$ pentru care $f(X, Y) \neq -f(V - X, Y)$.

27.1-6 Fiind dată o reţea de transport $G = (V, E)$ fie f_1 şi f_2 două funcţii din $V \times V$ în $\mathbb{R}$. ***Fluxul sumă*** $f_1 + f_2$ este funcţia din $V \times V$ în $\mathbb{R}$ definită prin

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \tag{27.4}$$

pentru orice $u, v \in V$. Dacă f_1 şi f_2 sunt fluxuri în G, care dintre cele trei proprietăţi ale fluxului trebuie să fie satisfăcută de $f_1 + f_2$ şi care poate fi încălcată?

27.1-7 Fie f un flux într-o reţea de transport, şi fie α un număr real. ***Produsul scalar de flux*** αf este o funcţie din $V \times V$ în $\mathbb{R}$ definită prin

$$(\alpha f)(u, v) = \alpha \cdot f(u, v).$$

Demonstraţi că fluxurile într-o reţea de transport formează o mulţime convexă, arătând că dacă f_1 şi f_2 sunt fluxuri, atunci şi $\alpha f_1 + (1 - \alpha) f_2$ este flux pentru orice α pentru care $0 \leq \alpha \leq 1$.

27.1-8 Formulaţi problema fluxului maxim sub forma unei probleme de programare liniară.

27.1-9 Modelul de reţea de transport prezentat în această secţiune permite lucrul cu fluxuri cu un singur tip de produs. O ***reţea de transport multiprodus*** permite lucrul cu fluxuri cu p produse între o mulţime de p ***vârfuri sursă*** $S = \{s_1, s_2, \ldots, s_p\}$ şi o mulţime de p ***vârfuri destinaţie*** $T = \{t_1, t_2, \ldots, t_p\}$. Fluxul celui de al i-lea produs de la u la v se notează prin $f_i(u, v)$. Pentru produsul al i-lea singura sursă este s_i şi singura destinaţie este t_i. Proprietatea de conservare este valabilă independent pentru fiecare produs în parte: fluxul pentru fiecare produs este zero în fiecare vârf diferit de sursa şi destinaţia produsului respectiv. Suma fluxurilor de la u la v pentru toate produsele nu poate depăşi valoarea de capacitate $c(u, v)$, fluxurile pentru diferitele produse interacţionează numai în acest sens. ***Valoarea*** fluxului pentru fiecare produs este fluxul în vârful sursă pentru produsul respectiv. ***Valoarea fluxului total*** este suma valorilor pentru cele p fluxuri. Demonstraţi că există un algoritm polinomial care rezolvă problema găsirii valorii fluxului total maxim într-o reţea de transport multiprodus prin formularea problemei ca o problemă de programare liniară.

27.2. Metoda lui Ford-Fulkerson

Această secţiune prezintă metoda lui Ford-Fulkerson pentru rezolvarea problemei de flux maxim. Vorbim mai degrabă de "metodă" decât de "algoritm" deoarece ea cuprinde mai multe implementări cu diferiţi timpi de execuţie. Metoda lui Ford-Fulkerson se bazează pe trei idei importante, care depăşesc cadrul metodei şi sunt utilizate şi în multe alte probleme legate de fluxuri: reţele reziduale, drumuri de ameliorare şi tăieturi. Aceste idei sunt esenţiale în demonstrarea teoremei de flux maxim–tăietură minimă (teorema 27.7), care caracterizează valoarea fluxului maxim cu ajutorul tăieturilor în reţele de transport. La sfârşitul acestei secţiuni vom prezenta o implementare a metodei lui Ford-Fulkerson şi vom analiza timpul ei de execuţie.

Metoda lui Ford-Fulkerson este iterativă. Vom începe cu un flux $f(u, v) = 0$ pentru orice $u, v \in V$, ca un flux iniţial de valoare 0. La fiecare pas al iteraţiei vom mări valoarea fluxului prin găsirea unui "drum de ameliorare", care este un drum de-alungul căruia se poate mări fluxul, deci şi valoarea lui. Vom repeta aceşti paşi până când nu se mai găseşte nici un drum de ameliorare. Teorema de flux maxim–tăietură minimă va demonstra că în acest caz obţinem fluxul maxim.

METODA-FORD-FULKERSON
1: fie f fluxul identic 0
2: **cât timp** există un drum p de ameliorare **execută**
3: măreşte fluxul f de-a lungul drumului p
4: **returnează** f

Reţele reziduale

Dacă se dă o reţea de transport şi un flux, se poate spune intuitiv că reţeaua reziduală constă din arcele care admit flux mai mare. Mai formal, să presupunem că avem o reţea de transport $G = (V, U)$ cu sursa s şi destinaţia t. Fie f un flux în G şi să considerăm o pereche de vârfuri $u, v \in V$. Cantitatea de flux *adiţional* care poate fi transportat de la u la v, fără a depăşi capacitatea $c(u, v)$, este ***capacitatea reziduală*** a arcului (u, v) definită prin

$$c_f(u, v) = c(u, v) - f(u, v). \tag{27.5}$$

De exemplu, dacă $c(u, v) = 16$ şi $f(u, v) = 11$ vom putea transporta în plus $c_f(u, v) = 5$ unităţi de produs fără a depăşi capacitatea dată a arcului (u, v). Dacă fluxul $f(u, v)$ este negativ, capacitatea reziduală $c_f(u, v)$ este mai mare dacât capacitatea $c(u, v)$. De exemplu, dacă $c(u, v) = 16$ şi $f(u, v) = -4$, atunci capacitatea reziduală $c_f(u, v)$ este egală cu 20. Vom putea interpreta această situaţie în felul următor. Există un flux de 4 unităţi de la v la u, care poate fi anulat prin 4 unităţi de flux de la u la v. Apoi, se pot transporta alte 16 unităţi de la u la v fără a încălca restricţia de capacitate pe arcul (u, v). Deci am transportat 20 unităţi, pornind de la un flux $f(u, v) = -4$, fără a depăşi capacitatea arcului.

Fiind dată o reţea de transport $G = (V, E)$ şi un flux f, ***reţeaua reziduală*** a lui G indus de f este $G_f = (V, E_f)$, unde

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

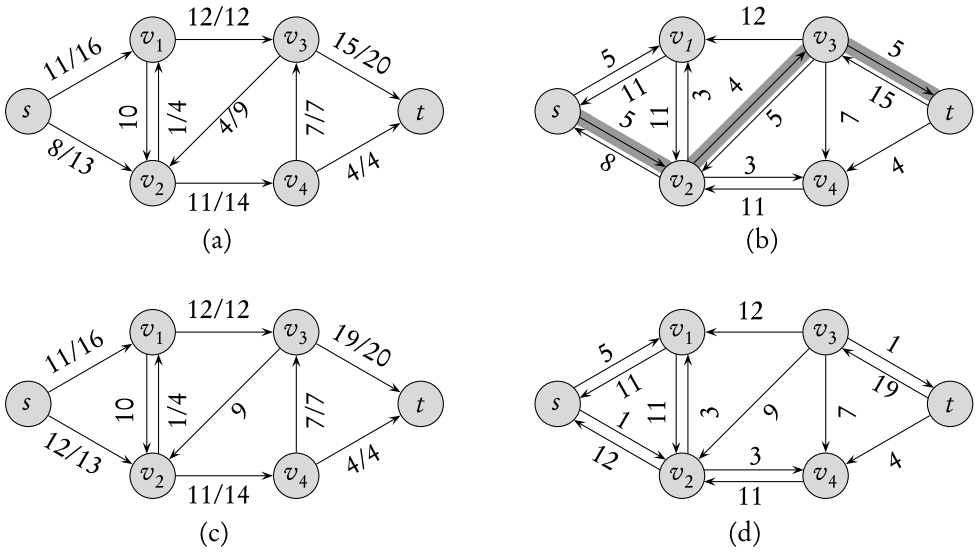


Figura 27.4 (**a**) Reţeaua de transport G şi fluxul f din figura 27.1(b) (**b**) Reţeaua reziduală G_f cu drumul de ameliorare p haşurat; capacitatea lui reziduală este $c_f(p) = c(v_2, v_3) = 4$. (**c**) Fluxul în G care rezultă prin mărirea fluxului de-a lungul drumului p având capacitatea reziduală 4. (**d**) Reţeaua reziduală indusă de fluxul din (c).

Deci, fiecare arc al reţelei reziduale, sau ***arc rezidual***, admite o creştere de flux strict pozitiv. În figura 27.4(a) reluăm reţeaua G şi fluxul f din figura 27.1(b), iar figura 27.4(b) ne arată reţeaua reziduală G_f.

Să observăm că (u, v) poate fi un arc rezidual în E_f chiar dacă el nu este arc în E. Altfel spus, se poate întâmpla ca $E_f \not\subseteq E$. Reţeaua reziduală din figura 27.4(b) include arce de acest fel care nu apar în reţeaua iniţială, cum ar fi de exemplu arcele (v_1, s) şi (v_2, v_3). Un astfel de arc (u, v) poate să apară în G_f numai dacă $(v, u) \in E$ şi există flux pozitiv de la v la u. Deoarece fluxul $f(u, v)$ de la u la v este negativ, $c_f(u, v) = c(u, v) - f(u, v)$ este pozitiv şi $(u, v) \in E_f$. Deoarece arcul (u, v) poate să apară în reţeaua reziduală numai dacă cel puţin unul din arcele (u, v) şi (v, u) apar în reţeaua originală, avem $|E_f| \leq 2|E|$.

Să observăm că însăşi reţeaua reziduală G_f este o reţea de transport cu funcţia de capacitate c_f. Lema următoare ne arată legătura dintr-un flux din reţeaua reziduală şi fluxul din reţeaua originală.

Lema 27.2 Fie $G = (V, E)$ o reţea cu vârful sursă s, respectiv vârful destinaţie t şi fie f un flux în G. Fie G_f reţeaua reziduală indusă de fluxul f şi fie f' un flux în G_f. Atunci fluxul sumă $f + f'$ definit prin formula (27.4) este un flux în G cu valoarea $|f + f'| = |f| + |f'|$.

Demonstraţie. Trebuie să verificăm că antisimetria, restricţia de capacitate şi proprietatea de conservare sunt verificate. Pentru a arăta antisimetria, să observăm că pentru orice $u, v \in V$ avem

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) &&= -f(v, u) - f'(v, u) \\ &= -(f(u, v) + f'(v, u)) &&= -(f + f')(v, u).\end{aligned}$$

Pentru restricţia de capacitate să observăm că $f'(u,v) \leq c_f(u,v)$ pentru orice $u, v \in V$. Ţinând cont de formula (27.5) avem

$$(f + f')(u,v) = f(u,v) + f'(u,v) \leq f(u,v) + (c(u,v) - f(u,v)) = c(u,v).$$

Pentru proprietatea de conservare avem, pentru orice $u \in V - \{s,t\}$

$$\sum_{v \in V} (f + f')(u,v) = \sum_{v \in V} f(u,v) + f'(u,v)) = \sum_{v \in V} f(u,v) + \sum_{v \in V} f'(u,v) = 0 + 0 = 0.$$

În final avem

$$\begin{aligned} |f + f'| &= \sum_{v \in V} (f + f')(s,v) = \sum_{v \in V} (f(s,v) + f'(s,v) \\ &= \sum_{v \in V} f(s,v) + \sum_{v \in V} f'(s,v) = |f| + |f'|. \end{aligned}$$

■

Drumuri de ameliorare

Dându-se o reţea de transport $G = (V,E)$ şi un flux f, un ***drum de ameliorare*** p este un drum simplu de la s la t în reţeaua reziduală G_f. După definiţia reţelei reziduale, fiecare arc (u,v) pe un drum de ameliorare admite un flux pozitiv adiţional, fără să încalce restricţia de capacitate.

Drumul haşurat în figura 27.4(b) este un drum de ameliorare. Tratând reţeaua reziduală G_f din figură ca o reţea de transport, vom putea transporta 4 unităţi de flux adiţional pe fiecare arc de-a lungul acestui drum de ameliorare, fără a încălca restricţia de capacitate, pentru că cea mai mică capacitate reziduală pe acest drum este $c_f(v_2, v_3) = 4$. Vom numi ***capacitate reziduală*** a lui p cantitatea maximă a fluxului care se poate transporta de-a lungul drumului de ameliorare p, dată de formula

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ este pe drumul } p\}.$$

Lema următoare, a cărei demonstraţie este lăsată ca un exerciţiu (vezi exerciţiul 27.2-7), fundamentează cele de mai sus.

Lema 27.3 Fie $G = (V,E)$ o reţea de transport, f un flux în G şi p un drum de ameliorare în G_f. Definim funcţia $f_p : V \times V \to \mathbb{R}$ prin

$$f_p(u,v) = \begin{cases} c_f(p) & \text{dacă } (u,v) \text{ este pe drumul } p, \\ -c_f(p) & \text{dacă } (v,u) \text{ este pe drumul } p, \\ 0 & \text{altfel.} \end{cases} \tag{27.6}$$

Atunci f_p este un flux în G_f cu valoarea $|f_p| = c_f(p) > 0$. ■

Corolarul următor ne arată că dacă adunăm f_p la f obţinem un alt flux în G a cărei valoare este mai aproape de valoarea maximă. În figura 27.4(c) se vede rezultatul adunării lui f_p din figura 27.4(b) la fluxul f din figura 27.4(a).

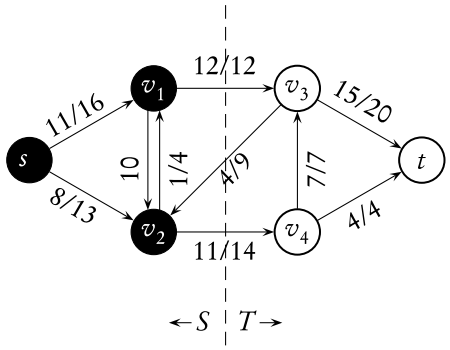


Figura 27.5 O tăietură (S, T) în reţeaua de transport din figura 27.1(b), unde $S = \{s, v_1, v_2\}$ şi $T = \{v_3, v_4, t\}$. Vârfurile din S sunt negre, iar cele din T sunt albe. Fluxul tăieturii (S, T) este $f(S, T) = 19$, iar capacitatea tăieturii $c(S, T) = 26$.

Corolarul 27.4 Fie $G = (V, E)$ o reţea de transport, f un flux în G şi p un drum de ameliorare în G_f. Fie funcţia f_p definită ca în formula (27.6). Definim funcţia $f' : V \times V \to \mathbb{R}$, $f' = f + f_p$. Atunci f' este un flux în G cu valoarea $|f'| = |f| + |f_p| > |f|$.

Demonstraţie. Demonstraţia este imediată pe baza lemelor 27.2 şi 27.3. ∎

Tăieturi în reţele de transport

Metoda lui Ford-Fulkerson măreşte fluxul în mod repetat de-a lungul drumurilor de ameliorare până când ajungem la un flux maxim. Teorema de flux maxim–tăietură minimă, pe care o vom demonstra pe scurt, ne exprimă faptul că un flux este maxim dacă şi numai dacă în reţeaua lui reziduală nu există nici un drum de ameliorare. Pentru a demonstra această teoremă trebuie să discutăm mai înainte noţiunea de tăietură într-o reţea de transport.

O ***tăietură*** (S, T) a unei reţele de transport $G = (V, E)$ este o partiţie a mulţimii V în mulţimile S şi $T = V - S$ astfel încât $s \in S$ şi $t \in T$. (Această definiţie seamănă cu definiţia "tăieturii" folosită pentru arborele parţial minim în capitolul 24, exceptând faptul că aici avem o tăietură într-un graf orientat şi nu neorientat ca acolo, şi că aici neapărat trebuie ca $s \in S$ şi $t \in T$.) Dacă f este un flux, atunci ***fluxul*** tăieturii (S, T) este definit ca fiind egal cu $f(S, T)$. De asemenea ***capacitatea*** tăieturii (S, T) este $c(S, T)$. O ***tăietură minimă*** este acea tăietură care are cea mai mică capacitate dintre toate tăieturile reţelei.

În figura 27.5 se poate vedea tăietura $(\{s, v_1 v_2\}, \{v_3, v_4, t\})$ al reţelei de transport din figura 27.1(b). Fluxul prin această tăietură este

$$f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) = 12 + (-4) + 11 = 19,$$

iar capacitatea ei este

$$c(v_1, v_3) + c(v_2, v_4) = 12 + 14 = 26.$$

Să observăm că fluxul unei tăieturi poate avea şi fluxuri negative între vârfuri, dar capacitatea unei tăieturi este compusă numai din valori nenegative.

După lema următoare valoarea unui flux într-o reţea este egală cu fluxul oricărei tăieturi.

Lema 27.5 Fie f un flux într-o reţea de transport G cu vârful sursă s şi vârful destinaţie t, şi fie (S,T) o tăietură în G. Atunci fluxul prin tăietura (S,T) este $f(S,T) = |f|$.

Demonstraţie. Din lema 27.1 rezultă

$$f(S,T) = f(S,V) - f(S,S) = f(S,V) = f(s,V) + f(S-s,V) = f(s,V) = |f|.$$

■

Un corolar imediat al lemei 27.5 este rezultatul deja demonstrat – formula (27.3) – şi anume că valoarea fluxului este fluxul în vârful sursă.

O altă consecinţă al lemei 27.5, exprimat în corolarul următor, ne spune cum se poate folosi capacitatea tăieturii pentru mărginirea valorii fluxului.

Corolarul 27.6 Valoarea oricărui flux f într-o reţea G este mărginită superior de capacitatea oricărei tăieturi în G.

Demonstraţie. Fie (S,T) o tăietură a reţelei G şi f un flux oarecare în G. Pe baza lemei 27.5 şi folosind resctricţia de capacitate, avem:

$$|f| = f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) \leq \sum_{u \in S} \sum_{v \in T} c(u,v) = c(S,T).$$

■

Avem acum toate informaţiile pentru a demonstra importanta teoremă de flux maxim-tăietură minimă.

Teorema 27.7 (Teorema de flux maxim-tăietură minimă) Dacă f este un flux în reţeaua $G = (V,E)$ cu vârful sursă s şi vârful destinaţie t, atunci următoarele condiţii sunt echivalente:
1. f este un flux maxim în G.
2. Reţeaua reziduală G_f nu conţine nici un drum de ameliorare.
3. Există o tăietură (S,T) în G pentru care $|f| = c(S,T)$.

Demonstraţie. (1) $\Rightarrow$ (2): Să presupunem prin absurd că f este un flux maxim în G şi totuşi G_f are un drum p de ameliorare. Atunci, conform corolarului 27.4 suma $f + f_p$, unde f_p este dat de formula (27.6) este un flux în G cu valoare strict mai mare decât $|f|$, contrazicând presupunerea că f este un flux maxim.

(2) $\Rightarrow$ (3): Să presupunem că G_f nu are nici un drum de ameliorare, adică în G_f nu există drum de la s la t. Să definim mulţimile S şi T în felul următor:

$$S = \{v \in V : \text{ există drum de la } s \text{ la } v \text{ în } G_f\}$$

şi $T = V - S$. Partiţia (S,T) este o tăietură: avem, evident, $s \in S$ şi $t \notin S$, pentru că nu există drum de la s la t în G_f. Pentru orice pereche de vârfuri u şi v astfel încât $u \in S$ şi $v \in T$, avem $f(u,v) = c(u,v)$, deoarece altfel am avea $(u,v) \in E_f$ şi deci $v \in S$. Conform lemei 27.5 avem atunci $|f| = f(S,T) = c(S,T)$.

(3) $\Rightarrow$ (1): Din corolarul 27.6 avem $|f| \leq c(S,T)$ pentru orice tăietură (S,T). Condiţia $|f| = c(S,T)$ implică atunci că f este un flux maxim. ■

Algoritmul de bază al lui Ford-Fulkerson

În fiecare iteraţie a metodei lui Ford-Fulkerson căutăm un drum *oarecare* de ameliorare p şi mărim fluxul f de-a lungul drumului p cu capacitatea reziduală $c_f(p)$. Următoarea implementare a metodei calculează fluxul maxim în graful $G = (V, E)$, actualizând fluxul $f[u, v]$ între oricare două vârfuri care sunt legate printr-un arc.[1] Dacă u şi v nu sunt legate printr-un arc în nici o direcţie, presupunem că $f[u, v] = 0$. Presupunem că valoarea capacităţii între vârfurile u şi v este dată de funcţia $c(u, v)$ calculabilă în timp constant şi că $c(u, v) = 0$ dacă $(u, v) \notin E$. (Într-o implementare obişnuită valorile $c(u, v)$ sunt păstrate în câmpurile unei liste de adiacenţă.) Capacitatea reziduală se calculează conform formulei (27.5). În descrierea algoritmului, $c_f(p)$ este variabila care păstrează capacitatea reziduală a drumului p.

FORD-FULKERSON(G, s, t)
1: **pentru** fiecare arc $(u, v) \in E[G]$ **execută**
2: $\quad f[u, v] \leftarrow 0$
3: $\quad f[v, u] \leftarrow 0$
4: **cât timp** există un drum p de la s la t în reţeaua reziduală G_f **execută**
5: $\quad c_f(p) \leftarrow \min\{c_f(u, v)$: (u, v) este pe drumul $p\}$
6: $\quad$ **pentru** fiecare arc (u, v) din p **execută**
7: $\quad\quad f[u, v] \leftarrow f[u, v] + c_f(p)$
8: $\quad\quad f[v, u] \leftarrow -f[v, u]$

Acest algoritm FORD-FULKERSON pur şi simplu detaliază pseudocodul METODA-FORD-FULKERSON descris la începutul secţiunii. Figura 27.6 ilustrează rezultatul fiecărei iteraţii la o execuţie a algoritmului. Liniile 1–3 iniţializează fluxul f cu valoarea 0. Ciclul **cât timp** din liniile 4–8 găseşte pe rând câte un drum de ameliorare p în G_f şi măreşte fluxul f de-a lungul lui p cu valoarea capacităţii reziduale $c_f(p)$. Când nu mai există drum de ameliorare atunci f este un flux maxim.

Analiza algoritmului FORD-FULKERSON

Timpul de execuţie a algoritmului FORD-FULKERSON depinde de modul de determinare (în linia 4) a drumului de ameliorare p. Dacă drumul este ales într-un mod nepotrivit, se poate întâmpla că algoritmul nu se opreşte: valoarea fluxului creşte succesiv, dar nu converge către valoarea maximă. Dacă însă drumul de ameliorare se alege folosind un algoritm de căutare în lăţime (secţiunea 23.2) atunci timpul de execuţie a algoritmului este polinomial. Înainte de a demonstra acest lucru vom da o margine superioară în cazul în care drumul este ales arbitrar şi toate capacităţile sunt întregi.

În practică problemele de flux maxim apar de obicei cu capacităţi care sunt numere întregi. Dacă capacităţile sunt numere raţionale, cu o scalare potrivită, ele pot fi transformate în numere întregi. Cu această presupunere, o implementare directă a algoritmului FORD-FULKERSON are un timp de execuţie $O(E|f^*|)$, unde f^* este fluxul maxim obţinut de algoritm. Timpul de execuţie al liniilor 1–3 este $\Theta(E)$. Ciclul **cât timp** din liniile 4–8 este executat de cel mult $|f^*|$ ori, deoarece valoarea fluxului creşte la fiecare pas cu cel puţin o unitate.

[1] Folosim paranteze drepte, când este de vorba de o variabilă de program (a cărei valoare se modifică) şi paranteze rotunde când este vorba de funcţie.

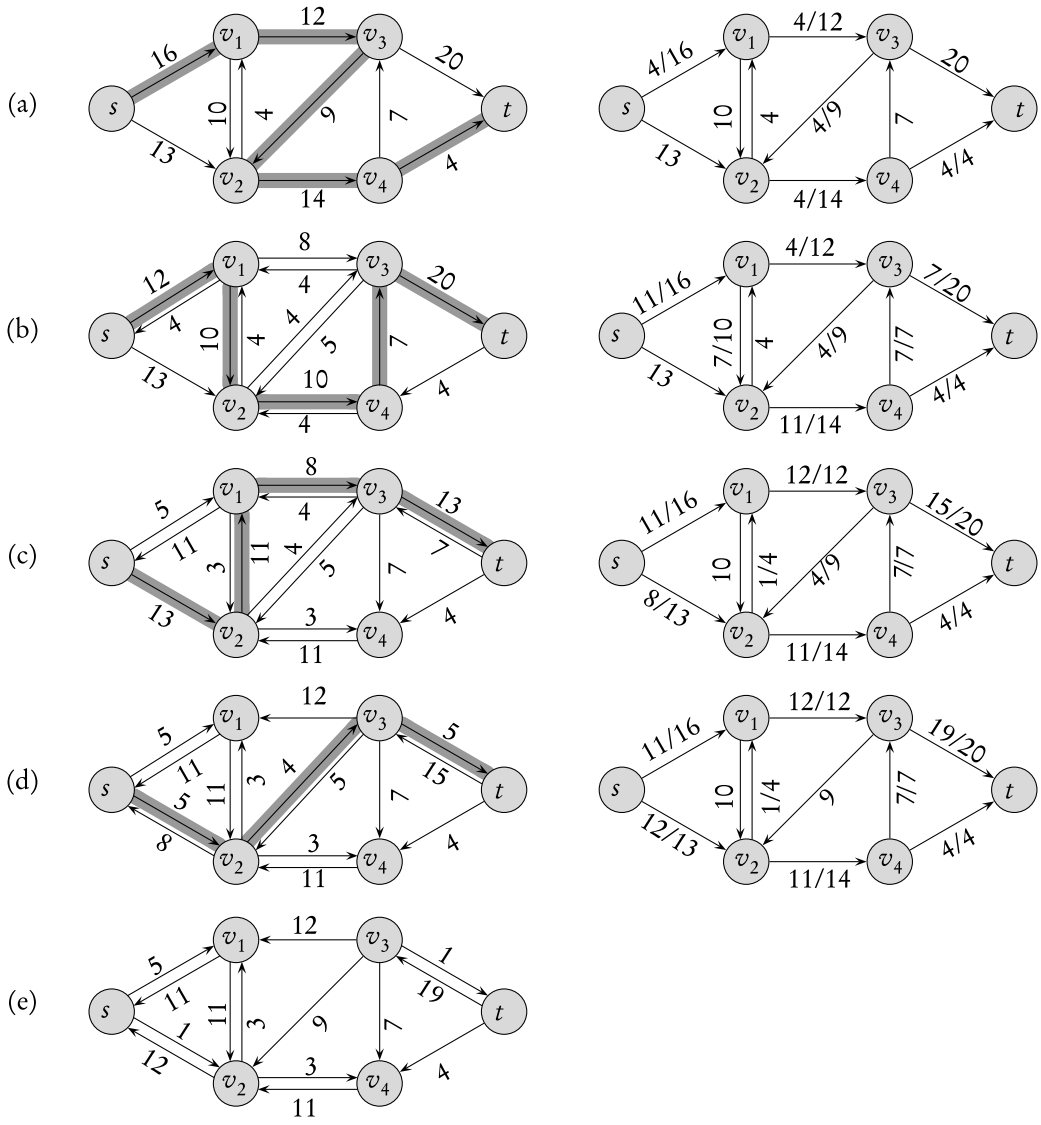


Figura 27.6 Execuţia algoritmului de bază al lui Ford-Fulkerson. (**a**)–(**d**) Iteraţii succesive ale ciclului **cât timp**. În partea stângă se află reţelele reziduale G_f din linia 4 al algoritmului; drumurile de ameliorare p sunt haşurate. În partea dreaptă se găsesc noile fluxuri f, rezultate prin adunarea lui f_p la f. Reţeaua reziduală din (a) este reţeaua de intrare G. (**e**) Reţeaua reziduală la ultimul test al ciclului **cât timp**. Această reţea nu mai are nici un drum de ameliorare, deci fluxul f de la (d) este un flux maxim.

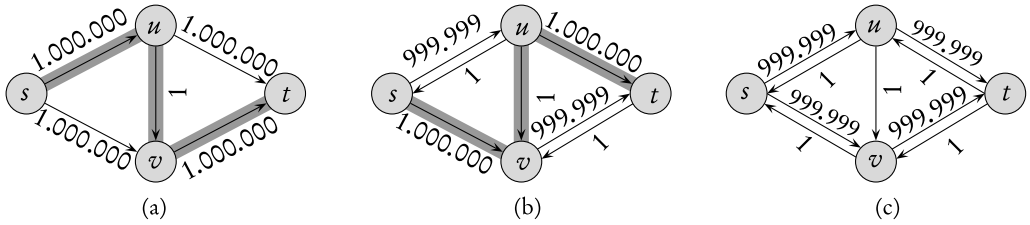


Figura 27.7 (**a**) O reţea de transport pentru care algoritmul FORD-FULKERSON are un timp de execuţie $\Theta(E|f^*|)$, unde f^* este fluxul maxim, aici $|f^*| = 2.000.000$. Se arată şi un drum de ameliorare cu capacitatea reziduală egală cu 1. (**b**) Reţeaua reziduală obţinută cu un alt drum de ameliorare având capacitatea reziduală egală cu 1. (**c**) Reţeaua reziduală obţinută.

Ciclul **cât timp** poate fi executat eficient dacă structura de date folosită la implementarea reţelei $G = (V, E)$ se gestionează eficient. Să presupunem că păstrăm o structură de date corespunzătoare unui graf orientat $G' = (V, E')$, unde $E' = \{(u, v) : (u, v) \in E \text{ sau } (v, u) \in E\}$. Arcele din G sunt arce şi în G', deci este uşor să se gestioneze capacităţile şi fluxurile într-o astfel de structură. Dându-se un flux f în G, arcele în reţeaua reziduală G_f costau din toate arcele (u, v) din G' pentru care $c(u, v) - f[u, v] \neq 0$. Timpul de găsire a unui drum în reţeaua reziduală este deci $O(E') = O(E)$, indiferent dacă folosim căutare în adâncime sau în lăţime. Fiecare iteraţie a ciclului **cât timp** se execută în $O(E)$ unităţi de timp, timpul total de execuţie a algoritmului FORD-FULKERSON este deci $O(E|f^*|)$.

Când capacităţile sunt numere întregi şi valoarea optimală $|f^*|$ a fluxului este mică, timpul de execuţie a algoritmului lui Ford-Fulkerson este bun. Figura 27.7(a) ne arată ce se poate întâmpla chiar într-o reţea simplă, dacă valoarea $|f^*|$ este mare. Un flux maxim în acest caz are valoarea 2.000.000. Pe drumul $s \to u \to t$ sunt transportate 1.000.000 unităţi, iar pe drumul $s \to v \to t$ alte 1.000.000 unităţi. Dacă primul drum de ameliorare ales de algoritmul FORD-FULKERSON este drumul $s \to u \to v \to t$, cum se arată în figura 27.7(a), valoarea fluxului după prima iteraţie este 1. Reţeaua reziduală corespunzătoare este dată în figura 27.7(b). Dacă în iteraţia a 2-a drumul de ameliorare ales este cel $s \to v \to u \to t$, aşa cum se arată în figura 27.7(b) atunci valoarea fluxului va fi 2. Figura 27.7(c) ne arată reţeaua reziduală rezultată. În continuare vom putea alege alternativ drumurile $s \to u \to v \to t$ respectiv $s \to v \to u \to t$. Vom avea astfel 2.000.000 de iteraţii, fluxul crescând la fiecare, cu câte o unitate.[2]

Performanţa algoritmului FORD-FULKERSON poate fi îmbunătăţită dacă implementăm căutarea drumului de ameliorare p printr-o căutare în lăţime, adică dacă drumul de ameliorare conţine cele mai *puţine* arce de la s la t (*cel mai scurt drum*, considerând că fiecare arc are ponderea 1). Vom numi această implementare a metodei lui Ford-Fulkerson ca fiind ***algoritmul lui Edmonds-Karp***. Vom demonstra în continuare că timpul de execuţie al algoritmului lui Edmonds-Karp este $O(VE^2)$.

Analiza algoritmului depinde de distanţa dintre vârfuri în reţeaua reziduală G_f. Lema următoare foloseşte notaţia $\delta_f(u, v)$ pentru cea mai scurtă distanţă de la vârful u la vârful v în G_f, unde fiecare arc are ponderea 1.

Lema 27.8 Dacă algoritmul lui Edmonds-Karp se foloseşte pentru o reţea de transport $G = (V, E)$ care are sursa s şi destinaţia t, atunci pentru orice vârf $v \in V - \{s, t\}$ distanţa $\delta_f(s, v)$ în

[2]Problema se poate rezolva în numai două iteraţii, dacă se aleg drumurile $s \to u \to t$ şi $s \to v \to t$ (n.t.).

reţeaua reziduală G_f creşte monoton cu fiecare ameliorare.

Demonstraţie. Presupunem prin absurd că pentru unele vârfuri $v \in V - \{s,t\}$ există mărire de flux pentru care $\delta_f(s,v)$ descreşte. Fie f fluxul înainte de mărire şi f' după. Atunci, conform presupunerii, avem

$$\delta_{f'}(s,v) < \delta_f(s,v).$$

Vom putea presupune, fără a pierde din generalitate, că $\delta_{f'}(s,v) \leq \delta_{f'}(s,u)$ pentru orice vârf $u \in V - \{s,t\}$ pentru care $\delta_{f'}(s,u) < \delta_f(u,v)$. Sau, ceea ce este acelaşi lucru, vom putea spune că

$$\delta_{f'}(s,u) < \delta_{f'}(s,v) \text{ implică } \delta_f(s,u) \leq \delta_{f'}(s,u). \tag{27.7}$$

Să considerăm acum un drum de lungime minimă (un cel mai scurt drum) p' în G_f de forma $s \rightsquigarrow u \rightarrow v$, unde u precede vârful v. Trebuie să avem $\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$ pe baza corolarului 25.2, deoarece (u,v) este un arc pe drumul p', care este un drum de lungime minimă de la s la v. Dar atunci din (27.7) avem

$$\delta_f(s,u) \leq \delta_{f'}(s,u).$$

Cu aceste vârfuri v şi u considerăm fluxul f de la u la v înainte de mărire a fluxului în G_f. Dacă $f[u,v] < c(u,v)$ avem

$$\begin{aligned} \delta_f(s,v) &\leq \delta_f(s,u) + 1 \qquad \text{(pe baza lemei 25.3)} \\ &\leq \delta_{f'}(s,u) + 1 = \delta_{f'}(s,v), \end{aligned}$$

ceea ce contrazice presupunerea că mărirea fluxului descreşte distanţa de la s la v.

Deci trebuie să avem $f[u,v] = c(u,v)$, ceea ce înseamnă că $(u,v) \notin E_f$. Atunci drumul de ameliorare p, ales în G_f pentru a produce $G_{f'}$ trebuie să conţină arcul (v,u) *orientat de la* v *la* u, deoarece $(u,v) \in E_{f'}$ (din ipoteză) şi tocmai am demonstrat că $(u,v) \notin E_f$. Deci, pe drumul de ameliorare p transportăm flux *înapoi* pe arcul (u,v) şi v apare înaintea lui u pe drumul p. Deoarece p este un drum de lumgime minimă de la s la t, toate subdrumurile lui (vezi lema 25.1) sunt de lungime minimă, şi deci vom avea $\delta_f(s,u) = \delta_f(s,v) + 1$. Deci

$$\delta_f(s,v) = \delta_f(s,u) - 1 \leq \delta_{f'}(s,u) - 1 = \delta_{f'}(s,v) - 2 < \delta_{f'}(s,v),$$

ceea ce contrazice ipoteza noastră iniţială. ∎

Următoarea teoremă ne dă o margine superioară pentru numărul de iteraţii în algoritmul lui Ednmonds-Karp.

Teorema 27.9 Dacă algoritmul lui Edmonds-Karp este aplicat pentru reţeaua de transport $G = (V,E)$ cu vârful sursă s şi destinaţie t, atunci fluxul se măreşte de cel mult $O(VE)$ ori.

Demonstraţie. Vom spune că un arc (u,v) în reţeaua reziduală G_f este un arc ***critic*** pe drumul de ameliorare p dacă capacitatea reziduală a drumului p este egală cu capacitatea reziduală a arcului (u,v), adică $c_f(p) = c_f(u,v)$. După mărirea fluxului de-a lungul drumului de ameliorarea toate arcele critice dispar din reţeaua reziduală. Mai mult, pe orice drum de ameliorarea trebuie să existe cel puţin un arc critic.

Fie u şi v două vârfuri din V legate printr-un arc din E. De câte ori poate fi critic acest arc (u, v) în timpul execuţiei algoritmului lui Edmonds-Karp? Pentru că drumul de ameliorare este un drum de lungime minimă, când arcul (u, v) este critic de prima dată, avem egalitatea

$$\delta_f(s, v) = \delta_f(s, u) + 1.$$

O dată ce fluxul este mărit, arcul (u, v) dispare din reţeaua reziduală. El nu poate să reapară pe un alt drum de ameliorare decât dacă fluxul pe arcul (u, v) descreşte, ceea ce se poate întâmpla numai dacă (v, u) apare pe un drum de ameliorare. Dacă f' este fluxul în G, atunci în acest caz

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1.$$

Deoarece $\delta_f(s, v) \leq \delta_{f'}(s, v)$ conform lemei 27.8 avem

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2.$$

În consecintă, din momentul în care arcul (u, v) devine critic până în momentul când a doua oară devine critic, distanţa lui u de la vârful sursă creşte cu cel puţin 2. Iniţial, distanţa lui u de la sursă este cel puţin 0, şi în momentul când nu mai poate fi atins din sursă (dacă acest lucru se întâmplă vreodată) distanţa lui va fi cel mult $|V| - 2$. Deci, arcul (u, v) poate deveni critic de cel mult $O(V)$ ori. Deoarece există $O(E)$ perechi de vârfuri care por fi legate prin arce în graful rezidual, numărul total de arce critice în timpul execuţiei algoritmului lui Edmonds-Karp este $O(VE)$. Deoarece fiecare drum de ameliorare are cel puţin un arc critic, teorema este demonstrată. ∎

Deoarece fiecare iteraţie în algoritmul FORD-FULKERSON poate fi implementată în $O(E)$ unităţi de timp, când drumul de ameliorare este găsit prin căutare în lăţime, timpul total de execuţie al algoritmului lui Edmonds-Karp este $O(VE^2)$. Algoritmul din secţiunea 27.4 ne va da o metodă cu timpul de execuţie $O(V^2E)$, care va fi baza algoritmului din secţiunea 27.5 cu timpul de execuţie $O(V^3)$.

Exerciţii

27.2-1 Cât este fluxul ce trece prin tăietura $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$ din figura 27.1(b)? Cât este capacitatea acestei tăieturi?

27.2-2 Care sunt paşii algoritmului lui Edmonds-Karp aplicat pentru reţeaua din figura 27.1(a)?

27.2-3 Care este tăietura minimă corespunzătoare fluxului din figura 27.6? Care două dintre drumurile de ameliorare care apar din exemplu micşorează fluxul pe vreun arc?

27.2-4 Demonstraţi că pentru orice pereche de vârfuri u şi v, şi pentru orice funcţie de capacitate c şi funcţie flux f avem $c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u)$.

27.2-5 După construcţia din secţiunea 27.1 o reţea cu mai multe surse şi destinaţii se poate transforma într-o reţea echivalentă cu o sursă unică şi destinaţie unică, adăugând arce având capacitate infinit. Demonstraţi că orice flux în reţeaua rezultată are o valoare finită dacă arcele în reţeaua originală au capacităţi finite.

27.2-6 Să presupunem că într-o problemă de flux cu surse multiple şi destinaţii multiple, fiecare sursă s_i produce p_i unităţi de flux, adică $f(s_i, V) = p_i$. Să presupunem de asemenea că fiecare destinaţie t_j consumă exact q_j unităţi de flux, adică $f(V, t_j) = q_j$, unde $\sum_i p_i = \sum_j q_j$. Arătaţi cum se poate transforma problema determinării fluxului f cu aceste condiţii suplimentare într-o problemă de flux într-o reţea cu o sursă unică şi destinaţie unică.

27.2-7 Demonstraţi lema 27.3.

27.2-8 Arătaţi că fluxul maxim într-o reţea $G = (V, E)$ totdeauna poate fi găsit printr-o secvenţă de cel mult $|E|$ drumuri de ameliorare. (*Indicaţie:* Determinaţi drumurile *după* obţinerea fluxului maxim.)

27.2-9 Conexiunea de muchii într-un graf neorientat este numărul minim k de muchii care trebuie eliminate pentru ca graful să devină neconex. De exemplu, conexiunea de muchii a unui arbore este 1, iar conexiunea de muchii a unui ciclu este 2. Arătaţi cum poate fi determinată conexiunea de muchii a unui graf neorientat $G = (V, E)$ aplicând un algoritm de flux maxim pe cel mult $|V|$ reţele de transport, fiecare având $O(V)$ vârfuri şi $O(E)$ muchii.

27.2-10 Presupunem că o reţea de transport $G = (V, E)$ are muchii simetrice, cu alte cuvinte $(u, v) \in E$ dacă şi numai dacă $(v, u) \in E$). Arătaţi că algoritmul lui Edmonds-Karp se termină după cel mult $|V||E|/4$ iteraţii. (*Indicaţie:* Verificaţi pentru orice arc (u, v) cum se schimbă valorile $\delta(s, u)$ şi $\delta(v, t)$ între două momente când arcul este critic.)

27.3. Cuplaj bipartit maxim

Anumite probleme de combinatorică pot fi considerate ca fiind probleme de flux maxim. Problema de flux maxim cu surse multiple şi destinaţii multiple din secţiunea 27.1 este un astfel de exemplu. Există şi alte probleme de combinatorică care, la prima vedere, nu par să aibe prea multe lucruri în comun cu reţele de transport, dar, de fapt, pot fi reduse la probleme de flux maxim. Acest paragraf prezintă o astfel de problemă: găsirea unui cuplaj maxim într-un graf bipartit (vezi secţiunea 5.4). Pentru a rezolva această problemă, vom folosi o proprietate de integralitate a metodei Ford-Fulkerson. De asemenea, vom vedea cum metoda Ford-Fulkerson poate fi folosită pentru a rezolva problema unui cuplaj bipartit maxim pentru un graf $G = (V, E)$ într-un timp $O(VE)$.

Problema cuplajului bipartit maxim

Fiind dat un graf neorientat $G = (V, E)$, un ***cuplaj*** este o submulţime de muchii $M \subseteq E$ astfel încât pentru toate vârfurile $v \in V$, există cel mult o muchie în M incidentă în v. Spunem că un vârf $v \in V$ este ***cuplat*** de cuplajul M dacă există o muchie în M incidentă în v; altfel, spunem că v este ***neconectată***. Un ***cuplaj maxim*** este un cuplaj de cardinalitate maximă, adică, un cuplaj M astfel încât pentru orice alt cuplaj M', avem $|M| \geq |M'|$. În acest paragraf, ne vom îndrepta atenţia asupra găsirii cuplajelor maxime în grafe bipartite. Vom presupune că mulţimea de vârfuri poate fi partiţionată în $V = L \cup R$, unde L şi R sunt disjuncte şi toate muchiile din E sunt între L şi R. Presupunem, de asemenea, că nu există vârfuri izolate în V. Figura 27.8 ilustrează noţiunea de cuplaj.

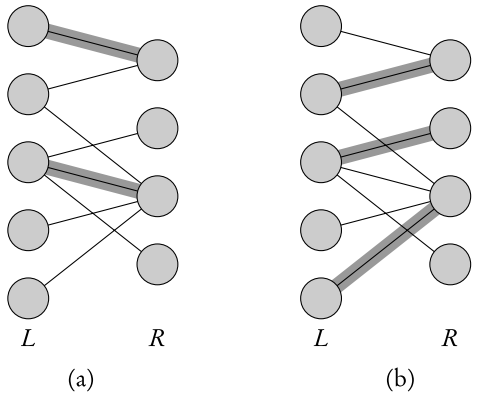


Figura 27.8 Un graf bipartit $G = (V, E)$ cu partiţia vârfurilor $V = L \cup R$. **(a)** Un cuplaj de cardinalitate 2. **(b)** Un cuplaj maxim de cardinalitate 3.

Problema găsirii unui cuplaj maxim într-un graf bipartit are multe aplicaţii practice. De exemplu, putem considera conectarea unei mulţimi de L maşini cu o mulţime de R task-uri care trebuie executate simultan. Vom considera că prezenţa muchiei (u, v) în E semnifică faptul că o anumită maşină $u \in L$ este capabilă să execute task-ul $v \in R$. Un cuplaj maxim va asigura funcţionarea a cât mai multor maşini posibile.

Determinarea cuplajului bipartit maxim

Putem folosi metoda Ford-Fulkerson pentru a determina cuplajul maxim într-un graf neorientat bipartit $G = (V, E)$, într-un timp polinomial în $|V|$ şi $|E|$. Soluţia constă în a construi o reţea de transport în care fluxurile reprezintă cuplaje, conform figurii 27.9. Vom defini ***reţeaua de transport corespunzătoare*** $G' = (V', E')$ pentru un graf bipartit G astfel. Vom alege sursa s şi destinaţia t ca fiind noi vârfuri care nu sunt în V şi vom construi $V' = V \cup \{s, t\}$. Dacă partiţia vârfurilor în G este $V = L \cup R$, atunci arcele orientate ale lui G' sunt date de:

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : u \in L, v \in R \text{ şi } (u, v) \in E\} \cup \{(v, t) : v \in R\} .$$

Pentru a completa construcţia, vom atribui fiecărei muchii din E' capacitatea unitate.

Teorema următoare exprimă faptul că un cuplaj în G corespunde în mod direct unui flux din reţeaua de transport a lui G'. Spunem că un flux f dintr-o reţea de transport $G = (V, E)$ este ***de valoare întreagă*** dacă $f(u, v)$ este un număr întreg pentru orice $(u, v) \in V \times V$.

Lema 27.10 Fie $G = (V, E)$ un graf bipartit cu partiţia vârfurilor $V = L \cup R$ şi fie $G' = (V', E')$ reţeaua de transport corespunzătoare acestui graf. Dacă M este un cuplaj în G, atunci există un flux f de valoare întreagă în G' cu valoarea $|f| = |M|$. Reciproc, dacă f este un flux de valoare întreagă în G', atunci există un cuplaj M în G de cardinalitate $|M| = |f|$.

Demonstraţie. Vom arăta prima dată că un cuplaj M în G corespunde unui flux de valoare întreagă în G'. Definim f astfel. Dacă $(u, v) \in M$ atunci $f(s, u) = f(u, v) = f(v, t) = 1$ şi $f(u, s) = f(v, u) = f(t, v) = -1$. Pentru toate celelalte arce $(u, v) \in E'$, vom defini $f(u, v) = 0$.

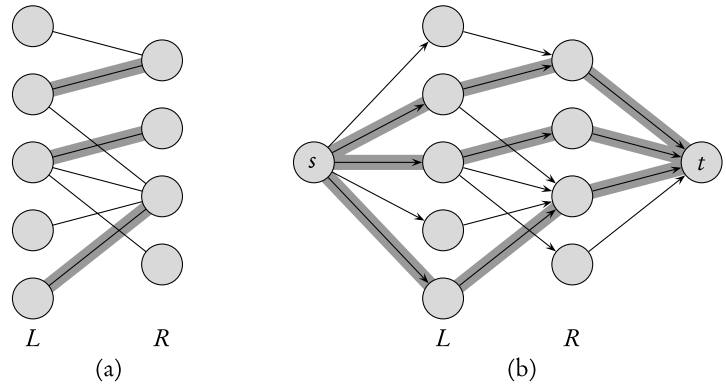


Figura 27.9 Reţeaua de transport corespunzătoare unui graf bipartit. **(a)** Graful bipartit $G = (V, E)$ cu partiţia vârfurilor $V = L \cup R$ din figura 27.8. Un cuplaj maxim este ilustrat prin muchii îngroşate. **(b)** Reţeaua de transport G' corespunzătoare, cu un flux maxim evidenţiat. Fiecare arc are capacitatea unitate. Arcele îngroşate au un flux egal cu 1 şi toate celelalte arce nu transportă nici un flux. Arcele îngroşate de la L la R corespund celor dintr-un cuplaj maxim al unui graf bipartit.

Intuitiv, fiecare muchie $(u, v) \in M$ corespunde unei unităţi de flux în G' care traversează drumul $s \to u \to v \to t$. Mai mult, drumurile determinate de muchii din M conţin vârfuri diferite, exceptând s şi t. Pentru a verifica faptul că într-adevăr f satisface simetria oblică, constrângerile de capacitate şi conservarea fluxului, trebuie doar să observăm că f poate fi obţinut prin mărirea fluxului de-a lungul fiecărui astfel de drum. Fluxul net de-a lungul tăieturii $L \cup \{s\}, R \cup \{t\}$ este egal cu $|M|$; astfel, conform lemei 27.5, valoarea fluxului este $|f| = |M|$.

Pentru a demonstra reciproca, fie f un flux de valoare întreagă în G' şi fie

$$M = \{(u, v) : u \in L, v \in R \text{ şi } f(u, v) > 0\}.$$

Fiecare vârf $u \in L$ are o singură muchie incidentă, fie ea (s, u) şi capacitatea ei este 1. Astfel, fiecare vârf $u \in L$ are cel mult o unitate de flux net pozitiv care intră în el. Deoarece f este de valoare întreagă, pentru fiecare $u \in L$, o unitate de flux net pozitiv intră în u dacă şi numai dacă există exact un vârf $v \in R$ astfel încât $f(u, v) = 1$. Deci, există cel mult o muchie care iese din fiecare vârf $u \in L$ care transportă flux net pozitiv. O argumentare similară poate fi făcută pentru fiecare vârf $v \in R$. Prin urmare, mulţimea M definită în enunţul teoremei este un cuplaj.

Pentru a arăta că $|M| = |f|$, să observăm că pentru fiecare vârf $u \in L$ conectat, avem $f(s, u) = 1$ şi pentru fiecare muchie $(u, v) \in E - M$ avem $f(u, v) = 0$. În consecinţă, folosind lema 27.1, simetria oblică şi faptul că nu există nici o muchie de la un vârf din L la t, obţinem:

$$|M| = f(L, R) = f(L, V') - f(L, L) - f(L, s) - f(L, t) = 0 - 0 + f(s, L) - 0 = f(s, V') = |f|.$$

■

Intuitiv putem spune că un cuplaj maxim într-un graf bipartit G corespunde unui flux maxim în reţeaua de transport G' corespunzătoare. Deci, putem calcula un cuplaj maxim în G printr-un algoritm de flux maxim în G'. Singurul dezavantaj al acestei abordări este că algoritmul de flux maxim poate produce ca rezultat un flux în G' care conţine valori care nu sunt întregi. Următoarea teoremă asigură faptul că dacă folosim metoda Ford-Fulkerson evităm această dificultate.

Teorema 27.11 (Teorema de integralitate) Dacă funcţia de capacitate c ia numai valori întregi, atunci fluxul maxim f produs de metoda Ford-Fulkerson are proprietatea că $|f|$ este de valoare întreagă. În plus, pentru toate vârfurile u şi v, valoarea $f(u, v)$ este un număr întreg.

Demonstraţie. Demonstraţia se face prin inducţie după numărul de iteraţii. O lăsăm pe seama cititorului în exerciţiul 27.3-2. ■

Putem acum demonstra următorul corolar al lemei 27.10.

Corolarul 27.12 Cardinalitatea unui cuplaj maxim într-un graf bipartit G este valoarea fluxului maxim din reţeaua de transport G' corespunzătoare.

Demonstraţie. Folosim noţiunile din lema 27.10. Presupunem că M este un cuplaj maxim în G şi că fluxul f corespunzător în G' nu este maxim. Atunci, există un flux maxim f' în G' astfel încât $|f'| > |f|$. Deoarece capacităţile din G' sunt valori întregi, conform teoremei 27.11 putem presupune că f' este de valoare întreagă. Prin urmare, f' corespunde unui cuplaj M' în G cu cardinalitatea $|M'| = |f'| > |f| = |M|$, contrazicând maximalitatea lui M. Similar, putem arăta că dacă f este un flux maxim în G', atunci cuplajul corespunzător este un cuplaj maxim în G. ■

În concluzie, fiind dat un graf G neorientat bipartit, putem determina un cuplaj maxim prin construcţia reţelei de transport G', apoi prin aplicarea metodei Ford-Fulkerson şi obţinând, în mod direct, un cuplaj maxim M dintr-un flux maxim f de valoare întreagă determinat. Deoarece orice cuplaj într-un graf bipartit are cardinalitatea cel mult $\min(L, R) = O(V)$, rezultă că valoarea fluxului maxim în G' este $O(V)$. Prin urmare, putem determina un cuplaj maxim într-un graf bipartit într-un timp $O(VE)$.

Exerciţii

27.3-1 Executaţi algoritmul Ford-Fulkerson pe reţeaua de transport din figura 27.9(b) şi ilustraţi reţeaua obţinută după fiecare mărire a fluxului. Numerotaţi vârfurile din L de sus în jos de la 1 la 5 şi în R de sus în jos de la 6 la 9. Pentru fiecare iteraţie, alegeţi drumul mărit care este cel mai mic în ordine lexicografică.

27.3-2 Demonstraţi teorema 27.11.

27.3-3 Fie $G = (V, E)$ un graf bipartit cu partiţia vârfurilor $V = L \cup R$ şi fie G' reţeaua de transport corespunzătoare. Determinaţi o margine superioară pentru lungimea oricărui drum mărit găsit în G' în timpul execuţiei lui FORD-FULKERSON.

27.3-4 ⋆ Un ***cuplaj perfect*** este un cuplaj în care fiecare vârf este cuplat. Fie $G = (V, E)$ un graf neorientat bipartit cu partiţia vârfurilor $V = L \cup R$, unde $|L| = |R|$. Pentru oricare $X \subseteq V$, definim ***vecinătatea*** lui X ca fiind

$$N(X) = \{y \in V : (x, y) \in E \text{ pentru } x \in X\},$$

adică mulţimea de vârfuri adiacente cu un anumit membru al lui X. Demonstraţi ***teorema lui Hall***: există un cuplaj perfect în G dacă şi numai dacă $|A| \leq |N(A)|$ pentru fiecare submulţime $A \subseteq L$.

27.3-5 ⋆ Un graf bipartit $G = (V, E)$, în care $V = L \cup R$, este ***d-regular*** dacă fiecare vârf $v \in V$ are exact gradul d. În oricare graf bipartit d-regular avem $|L| = |R|$. Demonstraţi că orice graf bipartit d-regular are un cuplaj de cardinalitate $|L|$, argumentând faptul că o tăietură minimă a reţelei de transport corespunzătoare are capacitatea $|L|$.

27.4. Algoritmi de preflux

În acest paragraf vom prezenta abordarea prin preflux a calculului fluxurilor maxime. Cei mai rapizi algoritmi de flux maxim până la ora actuală sunt algoritmii de preflux, cu ajutorul cărora se pot rezolva eficient şi alte probleme de flux, cum ar fi problema fluxului de cost minim. În acest paragraf vom introduce algoritmul "generic" de flux maxim al lui Goldberg, care are o implementare simplă ce se execută într-un timp $O(V^2E)$, îmbunătăţind astfel limita de $O(VE^2)$ al algoritmului Edmonds-Karp. Secţiunea 27.5 rafinează algoritmul generic pentru a obţine un alt algoritm de preflux cu timp de execuţie $O(V^3)$.

Algoritmii de preflux lucrează într-o manieră mai localizată decât metoda Ford-Fulkerson. În loc să examineze întreaga reţea rezultată $G = (V, E)$ pentru a determina drumul mărit, algoritmii de preflux acţionează asupra unui singur vârf la un moment dat, examinând doar vecinii vârfului din reţeaua rezultată. Mai mult, spre deosebire de metoda Ford-Fulkerson, algoritmii de preflux nu păstrează proprietatea de conservare a fluxului de-a lungul execuţiei lor. Totuşi, aceşti algoritmi gestionează un ***preflux***, care este o funcţie $f : V \times V \to \mathbb{R}$ care satisface simetria oblică, constrângerile de capacitate şi următoarea relaxare a conservării fluxului: $f(V, u) \geq 0$ pentru toare vârfurile $u \in V - \{s\}$. Această condiţie semnifică faptul că fluxul net în oricare vârf diferit de sursă este nenegativ. Numim ***exces de flux*** în u fluxul net în vârful u dat de

$$e(u) = f(V, u). \tag{27.8}$$

Spunem că un vârf $u \in V - \{s, t\}$ este ***cu exces de flux*** dacă $e(u) > 0$.

Vom începe acest paragraf prin a descrie intuitiv metoda de preflux. Vom studia apoi cele două operaţii implicate în această metodă: "pomparea prefluxului" şi "ridicarea" unui vârf. În final, vom prezenta algoritmul generic de preflux şi vom analiza corectitudinea şi timpul său de execuţie.

Intuiţia

Intuiţia din spatele metodei de preflux este, probabil, mai bine înţeleasă în termenii fluxurilor de fluide: considerăm o reţea de transport $G = (V, E)$ ca fiind un sistem de ţevi interconectate de capacităţi date. Aplicând această analogie metodei Ford-Fulkerson, putem spune că fiecare drum mărit în reţea produce o scurgere adiţională de fluid, fără puncte de ramificare, care curge de la sursă spre destinaţie. Metoda Ford-Fulkerson adaugă iterativ mai multe scurgeri de flux până când nu se mai pot adăuga alte scurgeri.

Algoritmul generic de preflux are o intuiţie oarecum diferită. Ca şi mai înainte, arcele orientate corespund unor ţevi. Vârfurile, care sunt îmbinări de ţevi, au două proprietăţi interesante. Prima, pentru a gestiona excesul de flux, constă în faptul că fiecare vârf are o ţeavă pentru scurgere într-un rezervor arbitrar suficient de mare care poate înmagazina fluid. A doua ne spune că,

fiecare vârf, de fapt, rezervorul corespunzător şi toate îmbinările sale cu alte ţevi se află pe o platformă a cărei înălţime creşte proporţional cu execuţia algoritmului.

Înălţimile vârfurilor determină modul în care este pompat fluxul: vom pompa fluxul doar de sus în jos, adică, de la un vârf mai înalt spre un vârf mai jos. Poate exista un flux net pozitiv de la un vârf mai jos spre un vârf mai sus, dar operaţiile care pompează fluxul vor împinge doar de sus în jos. Înălţimea sursei este fixată la $|V|$, iar înălţimea destinaţiei este 0. Toate celelalte înălţimi ale vârfurilor sunt iniţial 0 şi cresc în timp. La început, algoritmul trimite cât mai mult flux posibil de la sursă spre destinaţie. Cantitatea trimisă este exact cea necesară pentru a umple fiecare ţeavă ce pleacă de la sursă la întreaga capacitate; de fapt, trimite capacitatea tăietuii $(s, V - s)$. În momentul în care fluxul intră pentru prima dată într-un vârf intermediar, el este colectat în rezervorul vârfului. De acolo este, eventual, pompat în jos.

Este posibil ca singurele ţevi care pleacă din vârful u şi nu sunt încă saturate cu flux să fie conectate cu vârfuri situate pe acelaşi nivel ca şi u sau pe un nivel mai sus decât u. În acest caz, pentru a goli vârful u de excesul de flux, trebuie să-i mărim înălţimea – operaţie numită "ridicarea" vârfului u. Înălţimea sa este mărită cu o unitate faţă de înălţimea celui mai jos dintre vecinii săi, cu care este conectat printr-o ţeavă nesaturată. Prin urmare, după ce un vârf a fost ridicat, există cel puţin o ţeavă de ieşire prin care poate fi pompat mai mult flux.

În final, toate fluxurile care au putut trece prin destinaţie au ajuns la ea. Nu mai pot ajunge alte fluxuri, pentru că ţevile satisfac constrângerile de capacitate; cantitatea de flux de-a lungul oricărei tăieturi este în continuare limitată de capacitatea tăieturii. Pentru a transforma prefluxul într-un flux "valid", algoritmul trimite apoi excesul colectat în rezervoarele vârfurilor cu exces de flux înapoi la sursă, continuând să ridice vârfurile deasupra înălţimii fixate $|V|$ a sursei. (Transportul excesului înapoi la sursă este realizat, de fapt, prin anularea fluxurilor care au generat excesul). După cum vom vedea, în momentul în care toate rezervoarele au fost golite, prefluxul nu numai că este un flux "valid", el este totodată şi un flux maxim.

Operaţiile de bază

Din prezentarea anterioară observăm că există două operaţii de bază care se execută într-un algoritm de preflux: pomparea excesului de flux dintr-un vârf către unul din vecinii săi şi ridicarea unui vârf. Aplicabilitatea acestor operaţii depinde de înălţimile vârfurilor, pe care le vom defini acum formal.

Fie $G = (V, E)$ o reţea de transport cu sursa s şi destinaţia t şi fie f un preflux în G. O funcţie $h : V \leftarrow \mathbb{N}$ este o ***funcţie de înălţime*** dacă $h(s) = |V|, h(t) = 0$ şi

$$h(u) \leq h(v) + 1$$

pentru fiecare muchie rezultată $(u, v) \in E_f$. Obţinem următoarea lemă.

Lema 27.13 Fie $G = (V, E)$ o reţea de transport, fie f un preflux în G şi fie h o funcţie de înălţime pe V. Pentru oricare două vârfuri $u, v \in V$, dacă $h(u) > h(v) + 1$, atunci (u, v) nu este o muchie în graful rezultat. ∎

Operaţia de bază POMPEAZĂ(u, v) poate fi aplicată dacă u este un vârf cu exces de flux, $c_f(u, v) > 0$ şi $h(u) = h(v) + 1$. Pseudocodul următor actualizează prefluxul f într-o reţea $G = (V, E)$. Se presupune că sunt date capacităţile printr-o funcţie c constantă în timp şi că

fiind date c şi f, capacităţile rezultate se pot calcula în timp constant. Excesul de flux stocat în vârful u este gestionat de $e[u]$, iar înălţimea lui u este dată de $h[u]$. Expresia $d_f(u,v)$ este o variabilă temporară care memorează cantitatea de flux care poate fi pompată de la u la v.

POMPEAZĂ(u,v)
1: ▷ **Precondiţii**: u este excedentar, $c_f[u,v] > 0$ şi $h[u] = h[v] + 1$.
2: ▷ **Acţiune**: pompează $d_f(u,v) = \min(e[u], c_f(u,v))$ unităţi de flux de la u la v
3: $d_f(u,v) \leftarrow \min(e[u], c_f(u,v))$
4: $f[u,v] \leftarrow f[u,v] + d_f(u,v)$
5: $f[v,u] \leftarrow -f[u,v]$
6: $e[u] \leftarrow e[u] - d_f(u,v))$
7: $e[v] \leftarrow e[v] + d_f(u,v))$

Algoritmul POMPEAZĂ operează în felul următor: se presupune că vârful u are un exces pozitiv $e[u]$ şi capacitatea rezultată a lui (u,v) este pozitivă. Astfel, putem transporta până la $d_f(u,v) = \min(e[u], c_f(u,v))$ unităţi de flux de la u la v fără a cauza o valoare negativă pentru $e[u]$ şi fără a depăşi valoarea pentru capacitatea $c(u,v)$. Această valoare este calculată în linia 3. Fluxul este mutat de la u la v, actualizând valoarea lui f în liniile 4–5 şi valoarea lui e în liniile 6–7. Prin urmare, dacă f este un preflux înaintea apelului procedurii POMPEAZĂ, va rămâne un preflux şi după aceea.

După cum se observă, codul algoritmului POMPEAZĂ nu depinde în nici un moment de înălţimile lui u şi v, dar interzicem apelul ei dacă condiţia $h[u] = h[v] + 1$ nu este satisfăcută. Astfel, excesul de flux este pompat de sus în jos la o diferenţă de înălţime egală cu 1. Conform lemei 27.13 nu există muchii rezultate între două vârfuri cu diferenţă între înălţimi mai mare decât 1 şi, deci, nu se obţine nici un beneficiu permiţând fluxului să fie pompat de sus în jos la o diferenţă de înălţime mai mare decât 1.

Operaţia POMPEAZĂ(u,v) o vom numi ***pompare*** de la u la v. Dacă o operaţie de pompare este aplicată unei muchii (u,v) care pleacă dintr-un vârf u, putem spune şi că operaţia de pompare este aplicată lui u. O ***pompare saturată*** este cea prin care muchia (u,v) devine ***saturată*** (în continuare $c_f(u,v) = 0$), altfel este o ***pompare nesaturată***. Dacă o muchie nu este saturată atunci nu va apărea în reţeaua rezultată.

Operaţia de bază RIDICĂ(u) se aplică dacă u are exces de flux (este excedentar) şi dacă $c_f(u,v) > 0$ implică $h[u] \leq h[v]$ pentru toate vârfurile v. Cu alte cuvinte, putem ridica un vârf excedentar dacă pentru fiecare vârf v pentru care există o capacitate rezultată de la u la v, fluxul nu poate fi pompat de la u la v, deoarece v este situat mai jos decât u. (Să reamintim că, pe baza definiţiei, sursa s şi destinaţia t nu pot fi excedentare şi, deci, s şi t nu pot fi ridicate.)

RIDICĂ(u)
1: ▷ **Precondiţii**: u este excedentar şi pentru orice $v \in V$, $(u,v) \in E_f$ implică $h[u] \leq h[v]$.
2: ▷ **Acţiune**: Măreşte înălţimea lui u.
3: $h[u] \leftarrow 1 + \min\{h[v] : (u,v) \in E_f\}$

Când apelăm operaţia RIDICĂ(u), spunem că vârful u este ***ridicat***. Este important să menţionăm că atunci când u este ridicat, E_f trebuie să conţină cel puţin o muchie care iese din u, astfel ca minimul calculat în codul procedurii să corespundă unei mulţimi nevide. Această observaţie este dedusă din precondiţia ca u să fie excedentar. Deoarece $e[u] > 0$, avem

$e[u] = f[V, u] > 0$ şi, deci, trebuie să existe cel puţin un vârf v astfel încât $f[v, u] > 0$. Dar atunci

$$c_f(u, v) = c(u, v) - f[u, v] = c(u, v) + f[v, u] > 0,$$

ceea ce implică $(u, v) \in E_f$. Astfel, operaţia RIDICĂ(u) îi atribuie lui u cea mai mare înălţime permisă de constrângerile funcţiilor de înălţime.

Algoritmul generic

Algoritmul generic de preflux foloseşte următoarea subrutină pentru a crea prefluxul iniţial într-o reţea de transport.

INIŢIALIZEAZĂ-PREFLUX(G, s)
1: **pentru** fiecare vârf $u \in V[G]$ **execută**
2: $h[u] \leftarrow 0$
3: $e[u] \leftarrow 0$
4: **pentru** fiecare muchie $(u, v) \in E[G]$ **execută**
5: $f[u, v] \leftarrow 0$
6: $f[v, u] \leftarrow 0$
7: $h[s] \leftarrow V[G]$
8: **pentru** fiecare vârf $u \in Adj[s]$ **execută**
9: $f[s, u] \leftarrow c(s, u)$
10: $f[u, s] \leftarrow -c(s, u)$
11: $e[u] \leftarrow c(s, u)$

INIŢIALIZEAZĂ-PREFLUX creează un preflux iniţial f definit de:

$$f[u, v] = \begin{cases} c(u, v) & \text{dacă } u = s, \\ -c(v, u) & \text{dacă } v = s, \\ 0 & \text{altfel.} \end{cases}$$

Aceasta înseamnă că fiecare muchie care pleacă din vârful sursă este umplut la întreaga capacitate şi toate celelalte muchii nu transportă flux. Pentru fiecare vârf adiacent sursei avem iniţial $e[v] = c(s, v)$. De asemenea, algoritmul generic începe cu o funcţie de înălţime iniţială dată de

$$h[u] = \begin{cases} |V| & \text{dacă } u = s, \\ 0 & \text{altfel.} \end{cases}$$

Aceasta este o funcţie de înălţime pentru că singurele muchii (u, v) pentru care $h[u] > h[v]+1$ sunt cele pentru care $u = s$ şi aceste muchii sunt saturate, ceea ce înseamnă că nu vor fi în reţeua rezultată.

Următorul algoritm tipizează metoda de preflux.

PREFLUX-GENERIC(G)
1: INIŢIALIZEAZĂ-PREFLUX(G, s)
2: **cât timp** există o operaţie de pompare sau de ridicare ce se poate aplica **execută**
3: selectează o operaţie de pompare sau de ridicare ce se poate aplica şi execut-o

După iniţializarea fluxului, algoritmul generic aplică repetat, în orice ordine, operaţiile de bază ori de câte ori acest lucru este posibil. Următoarea lemă precizează faptul că atât timp cât există un vârf cu exces de flux, se poate aplica cel puţin una din cele două operaţii de bază.

Lema 27.14 (Un vârf excedentar poate fi pompat sau ridicat) Fie $G = (V, E)$ o reţea de transport cu sursa s şi destinaţia t, fie f un preflux şi fie h o funcţie arbitrară de înălţime pentru f. Dacă u este un vârf excedentar oarecare, atunci i se poate aplica fie o operaţie de pompare, fie una de ridicare.

Demonstraţie. Pentru orice muchie rezultată (u, v) avem $h(u) \leq h(v) + 1$ deoarece h este o funcţie de înălţime. Dacă lui u nu i se poate aplica o operaţie de pompare atunci pentru toate muchiile rezultate (u, v) trebuie să avem $h(u) < h(v) + 1$, ceea ce implică $h(u) \leq h(v)$. Prin urmare, lui u i se poate aplica o operaţie de ridicare. ∎

Corectitudinea metodei de preflux

Pentru a arăta că algoritmul generic de preflux rezolvă problema fluxului maxim, vom demonstra mai întâi că dacă se termină, prefluxul f este fluxul maxim. Apoi, vom demonstra că se termină. Vom începe cu unele observaţii relative la funcţia de înălţime h.

Lema 27.15 (Înălţimile vârfurilor nu scad niciodată) În timpul execuţiei algoritmului PREFLUX-GENERIC asupra unei reţele de transport $G = (V, E)$, înălţimile $h[u]$, pentru fiecare vârf $u \in V$, nu scad niciodată. Mai mult, ori de câte ori se aplică o operaţie de ridicare asupra unui vârf u, înălţimea sa $h[u]$ creşte cu cel puţin 1.

Demonstraţie. Deoarece înălţimile vârfurilor se modifică doar în timpul operaţiilor de ridicare, este suficient să demonstrăm a doua afirmaţie a lemei. Dacă vârful este ridicat, atunci pentru toate vârfurile v astfel încât $(u, v) \in E_f$, avem $h[u] \leq h[v]$; aceasta implică $h[u] < 1 + min\{h[v] : (u, v) \in E_f\}$ şi deci operaţia trebuie să mărească $h[u]$. ∎

Lema 27.16 Fie $G = (V, E)$ o reţea de transport cu sursa s şi destinaţia t. În timpul execuţiei algoritmului PREFLUX-GENERIC asupra lui G, atributul h este gestionat ca o funcţie de înălţime.

Demonstraţie. Demonstraţia se face prin inducţie după numărul de operaţii de bază executate. După cum am observat deja, iniţial, h este o funcţie de înălţime.

Cerem ca dacă h este o funcţie de înălţime, atunci operaţia RIDICĂ(u) să păstreze h ca funcţie de înălţime. Dacă studiem muchia rezultată $(u, v) \in E_f$ care pleacă din u, atunci operaţia RIDICĂ(u) asigură că $h[u] \leq h[v]+1$ după aplicarea ei. Să studiem acum muchia rezultată (w, u) care intră în vârful u. Conform lemei 27.15, $h[w] \leq h[u]+1$ înaintea aplicării operaţiei RIDICĂ(u), implică $h[w] < h[u] + 1$ după aplicarea ei. Deci, operaţia RIDICĂ(u) păstrează h ca funcţie de înălţime.

Să considerăm acum operaţia POMPEAZĂ(u, v). Această operaţie poate adăuga muchia (v, u) la E_f sau poate elimina (u, v) din E_f. În primul caz, avem $h[v] = h[u] - 1$ şi astfel h rămâne o funcţie de înălţime. În al doilea caz, eliminarea muchiei (u, v) din reţeaua rezultată conduce la eliminarea constrângerilor corespunzătoare şi, din nou, h rămâne o funcţie de înălţime. ∎

Următoarea lemă prezintă o proprietate importantă a funcţiilor de înălţime.

Lema 27.17 Fie $G = (V, E)$ o reţea de transport cu sursa s şi destinaţia t, fie f un preflux în G şi fie h o funcţie arbitrară de înălţime pe V. Atunci, nu există nici un drum de la sursa s la destinaţia t în reţeaua rezultată G_f.

Demonstraţie. Vom folosi metoda reducerii la absurd. Pentru aceasta, presupunem că există un drum $p = \langle v_0, v_1, ..., v_k \rangle$ de la s la t în G_f, unde $v_0 = s$ şi $v_k = t$. Fără a pierde din generalitate, putem presupune că p este un drum elementar şi deci $k < |V|$. Pentru $i = 0, 1, ..., k-1$, muchia $(v_i, v_{i+1}) \in E_f$. Deoarece h este o funcţie de înălţime, avem $h(v_i) \leq h(v_{i+1}) + 1$, pentru $i = 0, 1, ..., k-1$. Combinând aceste inegalităţi de-a lungul drumului p obţinem $h(s) \leq h(t) + k$. Dar, deoarece $h(t) = 0$, avem $h(s) \leq k < |V|$, ceea ce contrazice cerinţa ca $h(s) = |V|$ pentru o funcţie de înălţime. ∎

Suntem acum pregătiţi să arătăm că dacă algoritmul generic de preflux se termină, atunci prefluxul pe care îl calculează este un flux maxim.

Teorema 27.18 (Corectitudinea algoritmului generic preflux) Dacă execuţia algoritmului PREFLUX-GENERIC se termină, pentru o reţea de transport $G = (V, E)$ având sursa s şi destinaţia t, atunci prefluxul f calculat este un flux maxim pentru G.

Demonstraţie. Dacă algoritmul generic se termină, atunci fiecare vârf din $V - \{s, t\}$ trebuie să aibe un exces nul, datorită lemelor 27.14 şi 27.16 şi invariantului că f este tot timpul un preflux, deci nu există vârfuri excedentare. Prin urmare, f este un flux. Deoarece h este o funcţie de înălţime, conform lemei 27.17, nu există nici un drum de la s la t în reţeua rezultată G_f.Conform teoremei flux maxim-tăietură minimă, rezultă că f este un flux maxim. ∎

Analiza metodei de preflux

Pentru a arăta că într-adevăr algoritmul generic de preflux se termină, vom stabili o limită pentru numărul de operaţii pe care le execută. Fiecare dintre cele trei tipuri de operaţii – ridicări, pompări saturate şi pompări nesaturate – este mărginită separat. Cunoscând aceste limite, este uşor să construim un algoritm care se execută într-un timp $O(V^2E)$. Înaintea începerii analizei, vom demonstra o lemă importantă.

Lema 27.19 Fie $G = (V, E)$ o reţea de transport cu sursa s şi destinaţia t şi fie f un preflux în G. Atunci, pentru orice vârf excedentar u, există un drum elementar de la u la s în reţeaua rezultată G_f.

Demonstraţie. Folosim metoda reducerii la absurd. Fie $U = \{v : \text{există un drum elementar de la } u \text{ la } v \text{ în } G_f\}$ şi presupunem că $s \notin U$. Fie $\bar{U} = V - U$.

Presupunem că pentru fiecare pereche de vârfuri $v \in U$ şi $w \in \bar{U}$ avem $f(w, v) \leq 0$. De ce? Dacă $f(w, v) > 0$, atunci $f(v, w) < 0$, ceea ce implică $c_f(v, w) = c(v, w) - f(v, w) > 0$. Prin urmare, există o muchie $(v, w) \in E_f$ şi un drum elementar de forma $u \rightsquigarrow v \rightarrow w$ în G_f, ceea ce contrazice alegerea noastră relativă la w.

Astfel, trebuie să avem $f(\bar{U}, U) \leq 0$, deoarece fiecare termen din această sumă este nepozitiv. Astfel, din formula (27.8) şi lema 27.1, putem deduce că

$$e(U) = f(V, U) = f(\bar{U}, U) + f(U, U) = f(\bar{U}, U) \leq 0.$$

Excesele sunt nenegative pentru toate vârfurile din $V - \{s\}$; deoarece am presupus că $U \subseteq V - \{s\}$, trebuie să avem $e(v) = 0$ pentru toate vârfurile $v \in U$. În particular, $e(u) = 0$, ceea ce contrazice presupunerea că u este excedentar. ∎

Următoarea lemă mărgineşte înălţimea vârfurilor şi corolarul ei stabileşte o margine pentru numărul total de operaţii de ridicare care sunt executate.

Lema 27.20 Fie $G = (V, E)$ o reţea de transport cu sursa s şi destinaţia t. În orice moment, în timpul execuţiei lui PREFLUX-GENERIC asupra lui G, avem $h[u] \leq 2|V| - 1$ pentru toate vârfurile $u \in V$.

Demonstraţie. Înălţimile sursei s şi destinaţiei t nu se modifică niciodată deoarece aceste vârfuri sunt prin definiţie neexcedentare. Deci, întotdeauna vom avea $h[s] = |V|$ şi $h[t] = 0$.

Datorită faptului că un vârf este ridicat doar atunci când este excedentar, putem considera orice vârf excedentar $u \in V - \{s, t\}$. Conform lemei 27.19, există un drum elementar p de la u la s în G_f. Fie $p = \langle v_0, v_1, ..., v_k \rangle$, unde $v_0 = u, v_k = s$ şi $k \leq |V| - 1$, deoarece p este elementar. Avem $(v_i, v_{i+1}) \in E_f$, pentru $i = 0, 1, ..., k - 1$ şi, prin urmare, conform lemei 27.16, $h[v_i] \leq h[v_{i+1} + 1$. Dezvoltând aceste inegalităţi peste drumul p obţinem $h[u] = h[v_0] \leq h[v_k] + k \leq h[s] + (|V| - 1) = 2|V| - 1$. ∎

Corolarul 27.21 (Limita pentru operaţii de ridicare) Fie $G = (V, E)$ o reţea de transport cu sursa s şi destinaţia t. Atunci, în timpul execuţiei algoritmului PREFLUX-GENERIC asupra lui G, numărul de operaţii de ridicare este cel mult $2|V| - 1$ pentru un vârf şi cel mult $(2|V| - 1)(|V| - 2) < 2|V|^2$ în total.

Demonstraţie. Doar vârfurile din $V - \{s, t\}$, în număr de $|V| - 2$, pot fi ridicate. Fie $u \in V - \{s, t\}$. Operaţia RIDICĂ(u) măreşte $h[u]$. Valoarea $h[u]$ este iniţial 0 şi, conform lemei 27.20, creşte până la cel mult $2|V| - 1$. Deci, fiecare vârf $u \in V - \{s, t\}$ este ridicat de cel mult $2|V| - 1$ ori şi numărul total de operaţii de ridicare executate este cel mult $(2|V| - 1)(|V| - 2) < 2|V|^2$. ∎

Lema 27.20 ne ajută să mărginim şi numărul de pompări saturate.

Lema 27.22 (Limita pentru pompări saturate) În timpul execuţiei algoritmului PREFLUX-GENERIC asupra oricărei reţele de transport $G = (V, E)$, numărul de pompări saturate este cel mult $2|V||E|$.

Demonstraţie. Pentru orice pereche de vârfuri $u, v \in V$, luăm în considerare pompările saturate de la u la v şi de la v la u. Dacă astfel de pompări există, cel puţin una dintre (u, v) şi (v, u) este o muchie în E. Să presupunem că a apărut o pompare saturată de la u la v. Pentru ca să fie posibil să apară altă pompare de la u la v, algoritmul trebuie mai întâi să pompeze fluxul de la v la u, ceea ce nu este posibil decât după ce $h[v]$ creşte cu cel puţin 2. Asemănător, $h[u]$ trebuie să crească cu cel puţin 2 între două pompări saturate de la v la u.

Considerăm secvenţa A de numere întregi date de $h[u] + h[v]$ pentru fiecare pompare saturată care apare între vârfurile u şi v. Dorim să mărginim lungimea acestei secvenţe. Când apare prima pompare în oricare dintre direcţii între u şi v, trebuie să avem $h[u] + h[v] \geq 1$; deci primul întreg din A este cel puţin 1. Când apare ultima astfel de pompare, avem $h[u] + h[v] \leq (2|V| - 1) + (2|V| - 2) = 4|V| - 3$, conform lemei 27.20; prin urmare, ultimul întreg din A este cel mult $4|V| - 3$. Ţinând cont de argumentaţia din paragraful anterior, în A pot apărea cel mult toţi

ceilalţi întregi. În concluzie, numărul de întregi din A este cel mult $((4|V|-3)-1)/2+1 = 2|V|-1$. (Adăugăm 1 pentru a fi siguri că ambele capete ale secvenţei sunt numărate.) Numărul total de pompări saturate între vârfurile u şi v este cel mult $2|V|-1$. Înmulţind cu numărul total de muchii, obţinem că numărul total de pompări saturate poate fi cel mult $(2|V|-1)|E| < 2|V||E|$. ■

Următoarea lemă stabileşte o margine pentru numărul de pompări nesaturate din algoritmul generic de preflux.

Lema 27.23 (Limita pentru pompări nesaturate) În timpul execuţiei algoritmului PREFLUX-GENERIC asupra oricărei reţele de transport $G = (V, E)$, numărul de pompări nesaturate este cel mult $4|V|^2(|V| + |E|)$.

Demonstraţie. Definim o funcţie de potenţial $\Phi = \sum_{v \in X} h[v]]$, unde $X \subseteq V$ este o mulţime de vârfuri excedentare. Iniţial, $\Phi = 0$. Se observă că ridicarea unui vârf u măreşte Φ cu cel mult $2|V|$, deoarece mulţimea peste care se face însumarea este aceeaşi şi u nu poate fi ridicat mai mult decât înălţimea sa maximă posibilă, care, conform lemei 27.20, este cel mult $2|V|$. De asemenea, o pompare saturată de la vârful u la vârful v măreşte Φ cu cel mult $2|V|$, deoarece înălţimile nu se modifică şi doar vârful v, a cărui înălţime este cel mult $2|V|$, poate, eventual, deveni excedentar. În fine, să observăm că o pompare nesaturată de la u la v micşorează Φ cu cel puţin 1, deoarece u nu mai este excedentar după pompare, v devine excedentar după pompare chiar dacă nu era înaintea operaţiei, iar $h[v] - h[u] = -1$.

Deci, pe parcursul algoritmului, numărul total de creşteri în Φ este constrâns de corolarul 27.21 şi lema 27.22, la valoarea $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2(|V| + |E|)$. Deoarece $\Phi \geq 0$, rezultă că numărul total de descreşteri şi deci numărul total de pompări nesaturate este cel mult $4|V|^2(|V| + |E|)$. ■

Suntem acum capabili să realizăm analiza procedurii PREFLUX-GENERIC şi a oricărui algoritm care se bazează pe metoda de preflux.

Teorema 27.24 În timpul execuţiei algoritmului PREFLUX-GENERIC asupra oricărei reţele de transport $G = (V, E)$, numărul de operaţii de bază este $O(V^2E)$.

Demonstraţie. Pe baza corolarului 27.21 şi a lemelor 27.22 şi 27.23, demonstraţia este imediată. ■

Corolarul 27.25 Există o implementare a algoritmului generic de preflux care se execută într-un timp $O(V^2E)$ pentru orice reţea de transport $G = (V, E)$.

Demonstraţie. Cerinţa exerciţiului 27.4-1 este să demonstraţi cum se poate implementa algoritmul generic cu limite de $O(V)$ pentru operaţia de ridicare şi $O(1)$ pentru pompare. În aceste condiţii, demonstraţia corolarului este imediată.

Exerciţii

27.4-1 Arătaţi cum se poate implementa algoritmul generic de preflux, folosind un timp $O(V)$ pentru operaţia de ridicare şi $O(1)$ pentru pompare, pentru a obţine un timp total de $O(V^2E)$.

27.4-2 Arătaţi că algoritmul generic de preflux necesită un timp total de doar $O(VE)$ pentru a executa toate cele $O(V^2)$ operaţii de ridicare.

27.4-3 Presupunând că a fost determinat un flux maxim într-o reţea de transport $G = (V, E)$ folosind algoritmul de preflux, construiţi un algoritm rapid pentru a determina o tăietură minimă în G.

27.4-4 Construiţi un algoritm de preflux pentru a determina un cuplaj maxim într-un graf bipartit. Analizaţi acest algoritm.

27.4-5 Presupunând că toate capacităţile muchiilor dintr-o reţea de transport $G = (V, E)$ sunt din mulţimea $\{1, 2, ..., k\}$, analizaţi timpul de execuţie al algoritmului generic de preflux în funcţie de $|V|, |E|$ şi k. (*Indicaţie*: De câte ori se pot aplica pompări nesaturate asupra fiecărei muchii înainte ca ea să devină saturată?)

27.4-6 Arătaţi că linia 7 din algoritmul INIŢIALIZEAZĂ-PREFLUX poate fi modificată astfel:

$$h[s] \leftarrow |V[G]| - 2$$

fără a afecta corectitudinea, respectiv performanţa în cazul asimptotic pentru algoritmul generic de preflux.

27.4-7 Fie $\delta_f(u, v)$ distanţa (numărul de muchii) de la u la v în reţeaua rezultată G_f. Arătaţi că PREFLUX-GENERIC respectă proprietatea: $h[u] < |V|$ implică $h[u] \leq \delta_f(u, t)$ şi $h[u] \geq |V|$ implică $h[u] - |V| \leq \delta_f(u, s)$.

27.4-8 $\star$ La fel ca în exerciţiu anterior, $\delta_f(u, v)$ notează distanţa de la u la v în reţeaua rezultată G_f. Arătaţi cum se poate modifica algoritmul generic de preflux pentru a satisface următoarea proprietate: $h[u] < |V|$ implică $h[u] = \delta_f(u, t)$ şi $h[u] \geq |V|$ implică $h[u] - |V| = \delta_f(u, s)$. Timpul total necesar ca implementarea să satisfacă această cerinţă trebuie să fie $O(VE)$.

27.4-9 Arătaţi că numărul de pompări nesaturate executate de algoritmul PREFLUX-GENERIC pentru o reţea de transport $G = (V, E)$ este cel mult $4|V|^2|E|$ pentru $|V| \geq 4$.

27.5. Algoritmul mutare-în-faţă

Metoda prefluxului permite aplicarea operaţiilor de bază în absolut orice ordine. Se poate rezolva totuşi problema fluxului maxim mai repede decât marginea $O\left(V^2E\right)$ dată de corolarul 27.25 prin alegerea atentă a ordinii şi administrarea eficientă a structurii de date corespunzătoare reţelei. Vom examina acum algoritmul ***mutare-în-faţă***, care este un algoritm de preflux de complexitate temporală $O\left(V^3\right)$, care este asimptotic cel puţin la fel de bun ca şi $O(V^2E)$.

Algoritmul mutare-în-faţă gestionează o listă a vârfurilor reţelei. Algoritmul parcurge lista începând din faţă, selectând repetat un vârf u la care apare depăşire şi apoi "descărcându-l", adică realizând operaţiile de pompare şi ridicare până când u nu mai are un exces pozitiv. Indiferent care vârf este ridicat, el este mutat în capul listei (de aici şi numele de "mutare-în-faţă") după care algoritmul reia parcurgerea listei.

Corectitudinea şi analiza algoritmului mutare-în-faţă depind de noţiunea de muchii "admisibile": acele muchii din reţeaua reziduală prin care poate fi pompat fluxul. După demonstrarea unor proprietăţi despre reţeaua muchiilor admisibile, vom investiga operaţia de descărcare şi apoi vom prezenta analiza algoritmului mutare-în-faţă.

Muchii şi reţele admisibile

Dacă $G = (V, E)$ este o reţea de transport cu sursa s şi destinaţia t, f este un preflux în G, iar h este funcţia de înălţime, atunci spunem că (u, v) este o ***muchie admisibilă*** dacă $c_f(u, v) > 0$ şi $h(u) = h(v) + 1$. În caz contrar (u, v) este ***inadmisibilă***. ***Reţeaua admisibilă*** este $G_{f,h} = (V, E_{f,h})$, unde $E_{f,h}$ este mulţimea muchiilor admisibile.

Reţeaua admisibilă constă din acele muchii prin care poate fi pompat fluxul. Lema următoare demonstrează că această reţea este un graf orientat aciclic.

Lema 27.26 (Reţeaua admisibilă este aciclică) Dacă $G = (V, E)$ este o reţea de transport, f este un preflux din G şi h este funcţia de înălţime pe G, atunci reţeaua admisibilă $G_{f,h} = (V, E_{f,h})$ este aciclică.

Demonstraţie. Demonstraţia se va face prin reducere la absurd. Se presupune că $G_{f,h}$ conţine un ciclu $p = \langle v_0, v_1, \ldots, v_k \rangle$, unde $v_0 = v_k$ si $k > 0$. Întrucât fiecare muchie din p este admisibilă, avem $h(v_i - 1) = h(v_i) + 1$ pentru $i = 1, 2, \ldots, k$. Însumând de-a lungul ciclului obţinem

$$\sum_{i=1}^{k} h(v_i - 1) = \sum_{i=1}^{k} (h(v_i) + 1) = \sum_{i=1}^{k} h(v_i) + k.$$

Deoarece fiecare vârf din ciclul p apare o dată în fiecare sumă, rezultă că $0 = k$, ceea ce constituie o contradicţie. ∎

Următoarele două leme arată modul în care operaţiile de ridicare şi pompare schimbă reţeaua admisibilă.

Lema 27.27 Fie $G = (V, E)$ o reţea de transport, f un preflux în G şi h o funcţie de înălţime. Dacă la un vârf u apare depăşire şi (u, v) este o muchie admisibilă, atunci se aplică POMPEAZĂ(u, v). Operaţia nu produce nici o muchie admisibilă nouă, dar poate transforma muchia (u, v) într-una inadmisibilă.

Demonstraţie. Din definiţia unei muchii admisibile rezultă că fluxul poate fi pompat de la u la v. Întrucât u este de depăşire, se aplică operaţia POMPEAZĂ(u, v). Singura muchie reziduală nouă care poate fi creată prin pomparea fluxului de la u la v este muchia (v, u). Întrucât $h(v) = h(u) - 1$, muchia (v, u) nu poate deveni admisibilă. Dacă operaţia este de pompare saturată, atunci $c_f(u, v) = 0$ după care (u, v) devine inadmisibilă. ∎

Lema 27.28 Fie $G = (V, E)$ o reţea de transport, f un preflux din G şi h o funcţie de înălţime. Dacă un vârf u este de depăşire şi nu există muchii admisibile ce pleacă din u, atunci se aplică RIDICĂ(u). După o operaţia de ridicare există cel puţin o muchie admisibilă ce pleacă din u, dar nici o muchie admisibilă ce intră în u.

Demonstraţie. Dacă u este de depăşire, conform lemei 27.14 i se aplică o operaţie fie de pompare, fie de ridicare. Dacă nu există muchii admisibile ce pleacă din u, nu poate fi pompat nici un flux dinspre u şi se aplică operaţia RIDICĂ(u). După operaţia de ridicare avem $h[u] = 1 + \min\{h[v] : (u, v) \in E_f\}$. Astfel, dacă v este un vârf care realizează minimul în această mulţime, muchia (u, v) devine admisibilă. Deci, după ridicare există cel puţin o muchie admisibilă ce pleacă din u.

Pentru a arăta că nici o muchie admisibilă nu intră în u după o operaţie de ridicare, să presupunem că există un vârf v, astfel încât (v, u) să fie admisibilă. Atunci $h[v] = h[u] + 1$ după ridicare şi deci, $h[v] > h[u] + 1$ imediat înaintea ridicării. Dar din lema 27.13 rezultă că nu există nici o muchie reziduală între vârfuri a căror înălţime diferă cu mai mult de 1. Mai mult, ridicarea unui vârf nu schimbă reţeaua reziduală. Astfel (v, u) nu este în reţeaua reziduală şi deci nu poate fi în reţeaua admisibilă. ∎

Liste de adiacenţă

Muchiile dintr-un algoritm mutare-în-faţă sunt organizate în "liste de adiacenţă". Fiind dată o reţea de transport $G = (V, E)$, lista de adiacenţă $N[u]$ pentru vârful $u \in V$ este o listă simplu înlănţuită a vârfurilor adiacente lui u în G. Astfel, vârful v apare în lista $N[u]$ dacă $(u, v) \in E$ sau dacă $(v, u) \in E$. Lista de adiacenţă $N[u]$ conţine exact acele vârfuri v pentru care poate exista o muchie reziduală (u, v). Primul vârf din $N[u]$ este indicat de $cap[N[u]]$. Vârful ce urmează lui v în lista de adiacenţă este indicat de *urm-adiacent*$[v]$; acest pointer este NIL dacă v este ultimul vârf în lista de adiacenţă.

Algoritmul mutare-în-faţă ciclează prin fiecare listă de adiacenţă într-o ordine arbitrară fixată de execuţia algoritmului. Pentru fiecare vârf u, câmpul $actual[u]$ indică vârful curent luat în considerare din $N[u]$. La început $actual[u]$ este iniţializat cu $cap[N[u]]$.

Descărcarea unui vârf de depăşire

Un vârf de depăşire este ***descărcat*** prin pomparea întregului său exces de flux prin muchiile admisibile spre vârfurile adiacente, ridicând u cât este necesar pentru a obţine ca muchiile ce pleacă din u să devină admisibile. Pseudocodul este prezentat în continuare.

```
DESCARCĂ(u)
 1: cât timp e[u] > 0 execută
 2:    v ← actual[u]
 3:    dacă v = NIL atunci
 4:       RIDICĂ(u)
 5:       actual[u] ← cap[N[u]]
 6:    altfel
 7:       dacă c_f(u, v) > 0 şi h[u] = h[v] + 1 atunci
 8:          POMPEAZĂ(u, v)
 9:       altfel
10:          actual[u] ← urm-adiacent[v]
```

Figura 27.10 prezintă parcurgerea câtorva iteraţii ale ciclului **cât timp** din liniile 1–10 care se execută cât timp vârful u are un exces pozitiv. Fiecare iteraţie realizează exact una dintre cele trei acţiuni, depinzând de vârful actual v din lista de adiacenţă $N[u]$.

1. Dacă v este NIL, atunci am trecut de ultimul element din $N[u]$. Linia 4 ridică vârful u şi atunci linia 5 resetează vârful adiacent actual al lui u pentru a fi primul din $N[u]$. (Lema 27.29 afirmă că în această situaţie se aplică operaţia de ridicare.)

2. Dacă v nu este NIL şi (u, v) este o muchie admisibilă (determinată de testul din linia 7), atunci linia 8 pompează o parte (posibil şi integral) din excesul lui u spre vârful v.

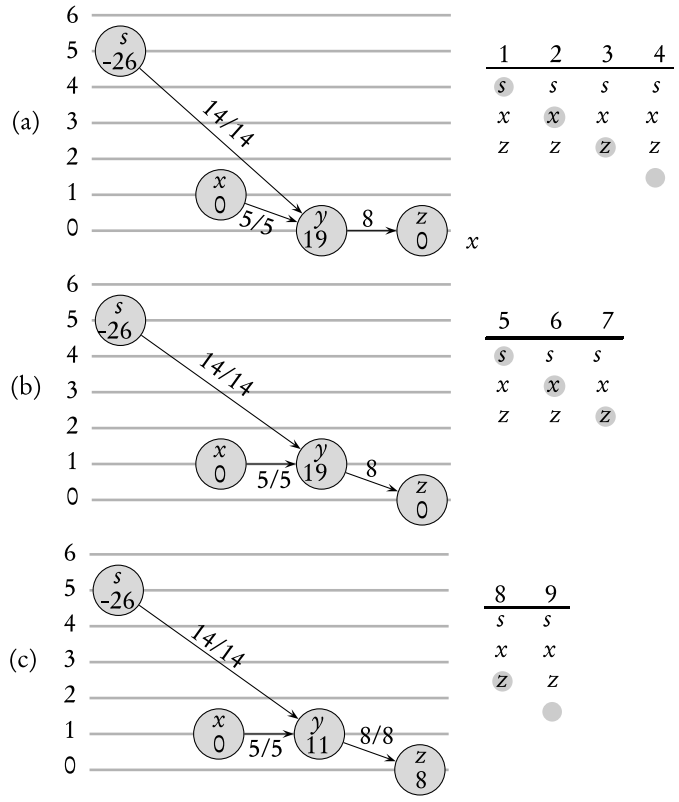
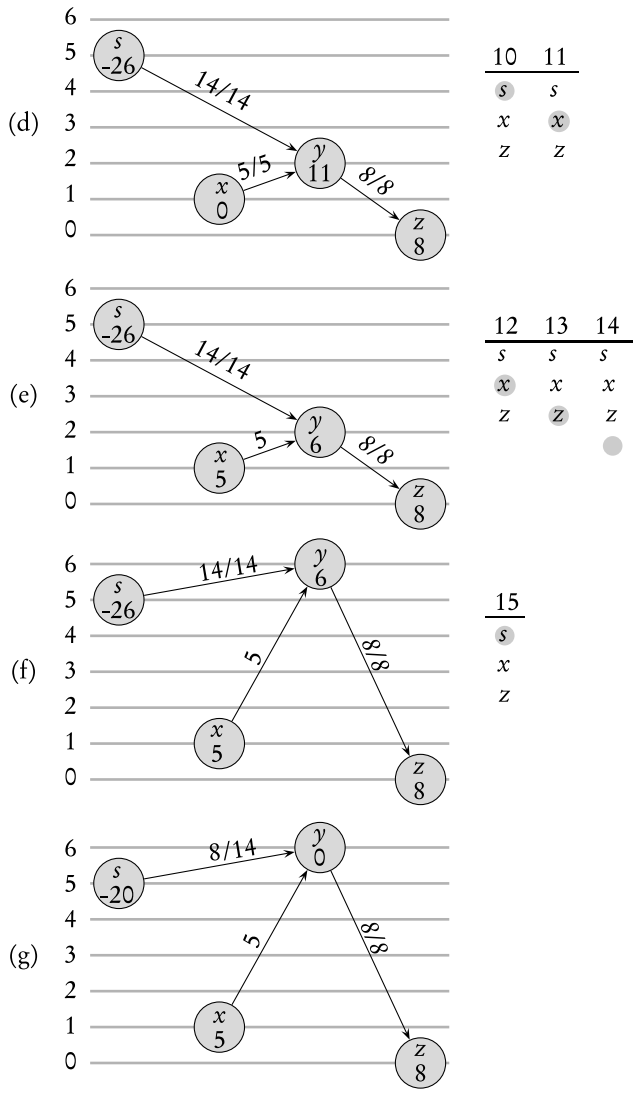


Figura 27.10 Descărcarea unui vârf. Necesită 15 iteraţii ale ciclului **cât timp** din DESCARCĂ pentru a pompa tot excesul de flux din vârful y. Doar vecinii lui y şi muchiile care intră în y sunt desenate. În fiecare componentă, numărul din interiorul fiecărui vârf este excesul său de la începutul fiecărei iteraţii desenate în componenta respectivă şi fiecare vârf este desenat la înălţimea componentei sale. La dreapta este lista de adiacenţă $N[y]$ la începutul fiecărei iteraţii, cu numărul iteraţiei sus. Vârful adiacent haşurat este $actual[y]$. **(a)** Iniţial sunt de pompat 19 unităţi de flux din y, iar $actual[y] = s$. Iteraţiile 1, 2 şi 3 doar îl avansează pe $actual[y]$, întrucât nu există muchii admisibile ce pleacă din y. În iteraţia 4 $actual[y] =$NIL (reprezentat haşurat) şi astfel y este ridicat, iar $actual[y]$ este resetat la capul listei de adiacenţă. **(b)** După ridicare vârful y are înălţimea 1. În iteraţiile 5, 6, muchiile (y, s) şi (y, x) sunt găsite a fi inadmisibile, dar 8 unităţi de flux în exces sunt pompate din y în z în iteraţia 7. Din cauza pompării, $actual[y]$ nu este avansat în această iteraţie. **(c)** Deoarece pomparea din iteraţia 7 a saturat muchia (y, z) ea este găsită inadmisibilă în saturaţia 8. În iteraţia 9 $actual[y] =$NIL şi astfel vârful y este din nou ridicat, iar $actual[y]$ resetat. **(d)** În iteraţia 10, (y, s) este inadmisibilă, dar în iteraţia 11 sunt pompate 5 unităţi de flux în exces din y în x. **(e)** Deoarece $actual[y]$ nu a fost avansat în iteraţia 11, iteraţia 12 găseşte (y, x) inadmisibilă. Iteraţia 13 găseşte (y, z) inadmisibilă, iar iteraţia 14 ridică vârful y şi resetează $actual[y]$. **(f)** Iteraţia 15 împinge 6 unităţi de flux în exces din y în s. **(g)** Vârful y nu mai are exces de flux şi DESCARCĂ se încheie. În acest exemplu DESCARCĂ începe şi se termină cu pointerul curent la capul listei de adiacenţă, dar în general nu este cazul.



3. Dacă v nu este NIL, dar (u, v) este inadmisibilă, atunci linia 10 ***avansează*** $actual[u]$ cu o poziţie mai departe în lista de adiacenţă $N[u]$.

De remarcat că dacă algoritmul DESCARCĂ este apelat pentru un vârf de depăşire u, atunci ultima acţiune realizată de DESCARCĂ trebuie să fie o pompare din u. De ce? Procedura se termină numai când $e[u]$ devine zero şi nici operaţia de ridicare, nici avansarea pointerului $actual[u]$ nu afectează valoarea lui $e[u]$.

Trebuie să ne asigurăm de faptul că atunci când POMPEAZĂ sau RIDICĂ este apelat de către DESCARCĂ, operaţia se aplică. Următoarea lemă va dovedi acest lucru.

Lema 27.29 Dacă DESCARCĂ apelează POMPEAZĂ(u, v) în linia 8, atunci o operaţie de pompare se aplică lui (u, v). Dacă DESCARCĂ apelează RIDICĂ(u) în linia 4, atunci lui u i se aplică o operaţie de ridicare.

Demonstraţie. Testele din liniile 1 şi 7 asigură că o operaţie de pompare apare numai dacă operaţia se aplică, ceea ce demonstrează prima afirmaţie din lemă.

Pentru a demonstra a doua afirmaţie, conform testului din linia 1 şi lemei 27.28, e nevoie să arătăm doar că toate muchiile care pleacă din u sunt inadmisibile. Se observă că pe măsură ce DESCARCĂ(u) este apelată repetat, pointerul $actual[u]$ se mişcă în josul listei $N[u]$. Fiecare "trecere" începe la capul lui $N[u]$ şi se termină cu $actual[u] =$NIL, moment în care u este ridicat şi începe o nouă trecere. Pentru ca pointerul $actual[u]$ să avanseze dincolo de un vârf $v \in N[u]$ în timpul unei treceri, muchia (u, v) trebuie să fie considerată inadmisibilă de către testul din linia 7. Astfel, în momentul în care trecerea se termină, fiecare muchie plecând din u a fost determinată ca fiind inadmisibilă la un moment dat în timpul trecerii. Observaţia cheie este că la sfârşitul trecerii, fiecare muchie care pleacă din u este încă inadmisibilă. De ce? Conform lemei 27.27 a pompării, nu pot crea nici o muchie admisibilă, lăsând una singură ce pleacă din u. Astfel, orice muchie admisibilă trebuie să fie întâi creată printr-o operaţie de ridicare. Dar vârful u nu este ridicat în timpul trecerii şi din lema 27.28 nici un alt vârf v care este ridicat în timpul trecerii nu are muchii admisibile ce intră în el. Astfel, la sfârşitul fiecărei treceri toate muchiile care pleacă din u rămân inadmisibile, deci lema este demonstrată. ∎

Algoritmul mutare-în-faţă

În algoritmul mutare-în-faţă gestionăm o listă înlănţuită L, formată din toate vârfurile din $V - \{s, t\}$. O proprietate cheie este că vârfurile din L sunt sortate topologic conform reţelei admisibile. (Amintim din lema 27.26 că reţeaua admisibilă este un graf orientat aciclic.)

Pseudocodul pentru algoritmul mutare-în-faţă presupune că listele de adiacenţă $N[u]$ au fost deja create pentru fiecare vârf u. El presupune şi că $urm[u]$ indică spre vârful care urmează lui u în lista L şi că $urm[u] =$NIL dacă u este ultimul vârf din listă.

```
MUTĂ-ÎN-FAŢĂ(G, s, t)
 1: INIŢIALIZEAZĂ-PREFLUX(G, s)
 2: L ← V[G] − {s, t}, în orice ordine
 3: pentru fiecare vârf u ∈ V[G] − {s, t} execută
 4:    actual[u] ← cap[N[u]]
 5: u ← cap[L]
 6: cât timp u ≠NIL execută
 7:    vechea-înălţime ← h[u]
 8:    DESCARCĂ(u)
 9:    dacă h[u] > vechea-înălţime atunci
10:       mută u în capul listei L
11:    u ← urm[u]
```

Algoritmul mutare-în-faţă lucrează după cum urmează. Linia 1 iniţializează prefluxul şi înălţimile la aceleaşi valori la fel ca şi algoritmul generic de preflux. Linia 2 iniţializează lista L astfel încât să conţină toate vârfurile potenţiale de depăşire, în orice ordine. Liniile 3–4 iniţializează pointerul curent al fiecărui vârf u spre primul vârf din lista de adiacenţă a lui u.

După cum se arată în figura 27.11, ciclul **cât timp** din liniile 6–11 parcurge lista L, descărcând vârfuri. Linia 5 ne asigură că parcurgerea începe cu primul vârf din listă. De fiecare dată de-a lungul ciclului este descărcat un vârf u în linia 8. Dacă u a fost ridicat de către procedura DESCARCĂ, linia 10 îl mută în capul listei L. Această determinare este făcută prin salvarea înălţimii lui u în variabila *vechea-înălţime* înainte de operaţia de descărcare (linia 7) şi compararea acestei înălţimi salvate cu înălţimea lui u de după aceea (linia 9). Linia 11 utilizează vârful următor lui u din lista L pentru a face următoarea iterare a ciclului **cât timp**. Dacă u a fost mutat în capul listei, vârful folosit în următoarea iteraţie este cel ce urmează lui u în noua sa poziţie din listă.

Pentru a arăta că MUTĂ-ÎN-FAŢĂ calculează un flux maxim, vom arăta că el este o implementare a algoritmului generic de preflux. Mai întâi, se observă că el realizează numai operaţie de pompare şi ridicare când ele se aplică, întrucât lema 27.29 garantează că DESCARCĂ le realizează numai când ele se aplică. Rămâne de arătat că atunci când MUTĂ-ÎN-FAŢĂ se termină, nu se aplică nici o operaţie de bază. Se observă că dacă u ajunge la sfârşitul lui L, fiecare vârf din L (cu posibila excepţie a primului vârf, care poate avea exces) trebuie descărcat fără a produce o ridicare. Lema 27.30, pe care o vom demonstra imediat, afirmă că lista L este gestionată ca o sortare topologică a reţelei admisibile. Astfel, o operaţie de pompare are ca efect mişcarea fluxului în exces spre vârfuri mai în josul listei (sau spre s sau t). Dacă pointerul u ajunge la sfârşitul listei, excesul fiecărui vârf este 0 şi nici una din operaţiile de bază nu se aplică.

Lema 27.30 Dacă executăm MUTĂ-ÎN-FAŢĂ pe o reţea de transport $G = (V, E)$ cu sursa s şi destinaţia t, atunci fiecare iteraţie a ciclului **cât timp** din liniile 6–11 menţine invariant faptul că L este o sortare topologică a vârfurilor din reţeaua admisibilă $G_{f,h} = (V, E_{f,h})$.

Demonstraţie. Imediat după ce INIŢIALIZEAZĂ-PREFLUX a fost executat, $h[s] = |V|$ şi $h[v] = 0$ pentru oricare $v \in V - \{s\}$. Întrucât $|V| \geq 2$ (deoarece el îl conţine cel puţin pe s şi t), nici o muchie nu poate fi admisibilă. Astfel $E_{f,h} = \emptyset$ şi orice ordonare a lui $V - \{s, t\}$ este o sortare topologică a lui $G_{f,h}$.

Vom demonstra acum că invariantul este menţinut prin fiecare iteraţie a ciclului **cât timp**. Reţeaua admisibilă este schimbată numai prin operaţii de pompare şi ridicare. Conform lemei 27.27 operaţiile de pompare produc numai muchii inadmisibile. Astfel, muchii admisibile pot fi create numai prin operaţii de ridicare. După ce un vârf este ridicat, totuşi, lema 27.28 afirmă că nu există muchii admisibile ce intră în u, dar pot există muchii admisibile ce pleacă din u. Astfel, prin deplasarea lui u în capul listei L, algoritmul asigură că orice muchie admisibilă plecând din u satisface ordonarea sortării topologice. ∎

Analiză

Vom arăta acum că timpul de execuţie a algoritmului MUTĂ-ÎN-FAŢĂ este $O(V^3)$ pe orice reţea de transport $G = (V, E)$. Întrucât algoritmul este o implementare a algoritmului de preflux, vom folosi corolarul 27.21, care furnizează o margine $O(V)$ pentru numărul de operaţii de ridicare executate pentru fiecare vârf şi o margine $O(V^2)$ pentru numărul total de ridicări. În plus, exerciţiul 7.4-2 furnizează o margine $O(VE)$ pentru timpul total consumat cu efectuarea operaţiilor de ridicare, iar lema 27.22 furnizează o margine $O(VE)$ pentru numărul total de operaţii de pompare saturate.

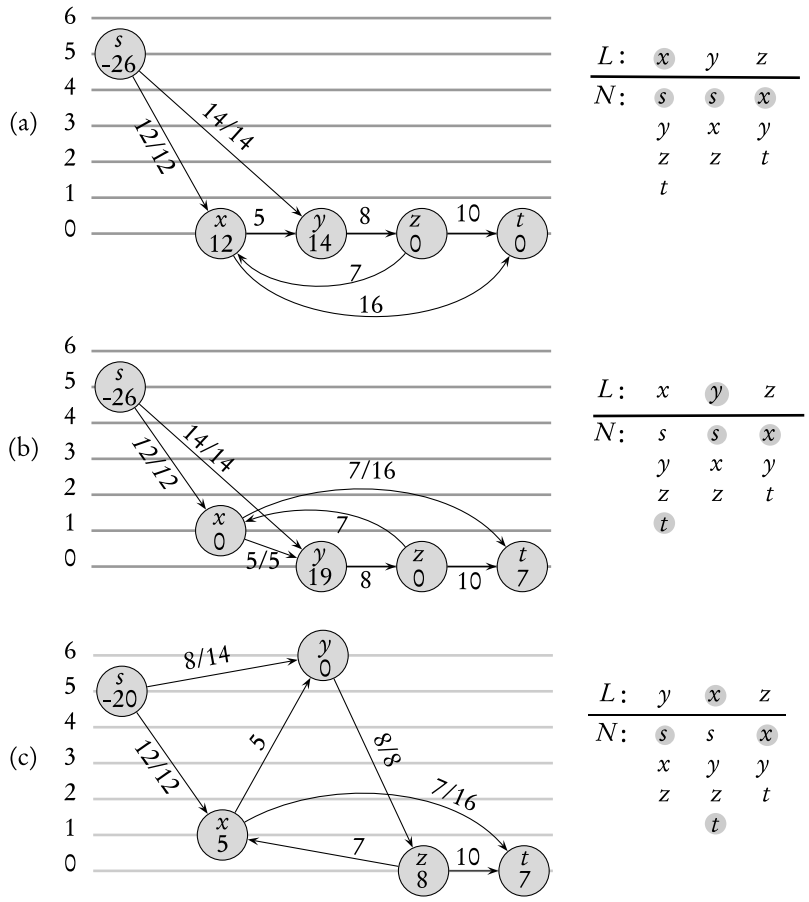
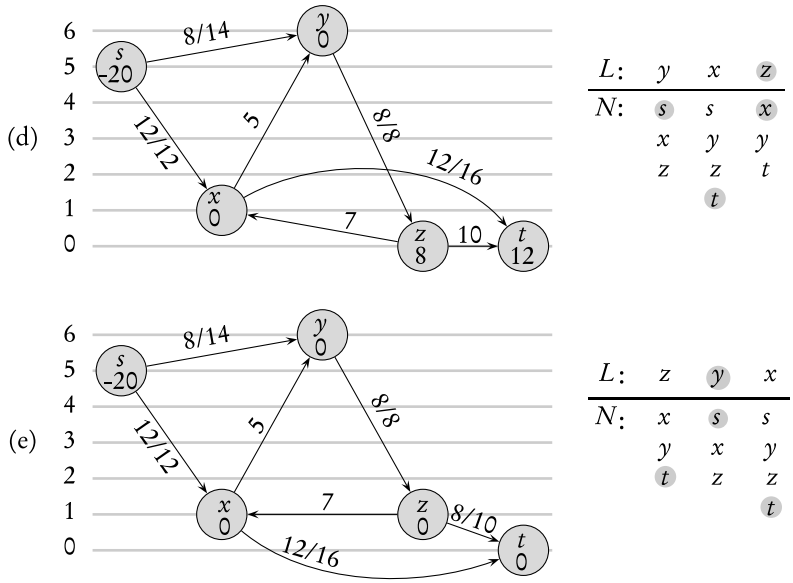


Figura 27.11 Acţiunea algoritmului MUTĂ-ÎN-FAŢĂ. **(a)** O reţea de transport imediat înainte de prima iteraţie a ciclului **cât timp**. Iniţial 26 de unităţi de flux părăsesc sursa s. La dreapta este dată lista iniţială, $L = \langle x, y, z \rangle$ cu $u = x$. Sub fiecare vârf din lista L este lista sa de adiacenţă, cu vecinul curent colorat cu negru. Vârful x este descărcat. El este ridicat la înălţimea 1, sunt pompate spre y 5 unităţi de flux, iar cele 7 unităţi rămase în exces sunt pompate spre destinaţia t. Deoarece x este ridicat, el este mutat în capul lui L, care în cazul acesta nu schimbă structura lui L. **(b)** Următorul vârf care este descărcat este y deoarece el urmează în L după x. Figura 27.10 arată acţiunea detaliată a descărcării lui y în această situaţie. Deoarece y este ridicat, el este mutat în capul lui L. **(c)** Vârful x îi urmează acum lui y în L şi astfel el este din nou descărcat, pompând toate cele 5 unităţi de flux în exces spre t. Deoarece vârful x nu este ridicat în această operaţie de descărcare, el rămâne pe loc în lista L. **(d)** Întrucât vârful z urmează după vârful x în L, el este descărcat. El este ridicat la înălţimea 1 şi toate cele 8 unităţi de flux în exces sunt pompate spre t. Deoarece z este ridicat, el este mutat în capul listei L. **(e)** Vârful y îi urmează acum lui z în L şi de aceea el este din nou descărcat. Deoarece y nu are exces, DESCARCĂ se întoarce imediat şi y rămâne pe loc în L. Vârful x este apoi descărcat. Deoarece nici el nu are exces, DESCARCĂ se întoarce din nou şi x rămâne pe loc în L. MUTĂ-ÎN-FAŢĂ a atins sfârşitul listei L şi se termină. Nu mai există vârfuri cu depăşire şi prefluxul este un flux maxim.



Teorema 27.31 Timpul de execuţie al algoritmului MUTĂ-ÎN-FAŢĂ pe orice reţea de transport $G = (V, E)$ este $O(V^3)$.

Demonstraţie. Să considerăm o "fază" a algoritmului mutare-în-faţă ca fiind durata dintre două operaţii consecutive de ridicare. Există $O(V^2)$ faze, deoarece sunt $O(V^2)$ operaţii de ridicare. Fiecare fază constă din cel mult $|V|$ apelări ale procedurii DESCARCĂ, după cum urmează. Dacă DESCARCĂ nu realizează o operaţie de ridicare, următorul apel al procedurii DESCARCĂ este mai în josul listei L şi lungimea lui L este mai mică decât $|V|$. Dacă DESCARCĂ realizează o ridicare, următoarea apelare a lui DESCARCĂ aparţine unei faze diferite. Deoarece fiecare fază conţine cel mult $|V|$ apelări ale lui DESCARCĂ şi sunt $O(V^2)$ faze, numărul de apeluri ale procedurii DESCARCĂ în linia 8 din procedura MUTĂ-ÎN-FAŢĂ este $O(V^3)$. Astfel, lucrul total realizat de ciclul **cât timp** în MUTĂ-ÎN-FAŢĂ, excluzând cel realizat în DESCARCĂ, este cel mult $O(V^3)$.

Trebuie acum să evaluăm volumul de calcul realizat de DESCARCĂ în timpul execuţiei algoritmului. Fiecare iteraţie a ciclului **cât timp** din DESCARCĂ realizează una dintre trei acţiuni. Vom analiza volumul total de calcul realizat de fiecare dintre aceste acţiuni.

Începem cu operaţiile de ridicare (liniile 4–5). Exerciţiul 27.4-2 furnizează o margine a timpului $O(VE)$ pentru toate cele $O(V^2)$ ridicări care sunt executate.

Să presupunem acum că acţiunea actualizează pointerul $actual[u]$ din linia 8. Această acţiune se produce de $O(\text{grad}(u))$ ori de câte ori un vârf u este ridicat şi de $O(V \cdot \text{grad}(u))$ ori global pentru acel vârf. Deci, pentru toate vârfurile, volumul total de calcul necesar pentru avansarea pointerilor în lista de adiacenţă este $O(VE)$ conform lemei strângerii de mână (exercitiul 5.4-1).

Al treilea tip de acţiune realizată de DESCARCĂ este o operaţie de pompare (linia 7). Ştim deja că numărul total de operaţii de pompare saturate este $O(VE)$. Se observă că dacă este executată o pompare nesaturată, DESCARCĂ se termină imediat, întrucât pomparea reduce excesul la 0. Astfel, există cel mult o pompare nesaturantă pentru fiecare apel al procedurii DESCARCĂ. După cum am observat, DESCARCĂ este apelată de $O(V^3)$ ori şi astfel timpul total consumat realizând pompări nesaturate este $O(V^3)$.

Timpul de execuţie al algoritmului MUTĂ-ÎN-FAŢĂ este deci $O(V^3 + VE)$, care este $O(V^3)$. ■

Exerciţii

27.5-1 Ilustraţi execuţia algoritmului MUTĂ-ÎN-FAŢĂ în stilul figurii 27.11 pentru reţeaua de transport din figura 27.1(a). Presupuneţi că ordinea iniţială a vârfurilor în lista L este $\langle v_1, v_2, v_3, v_4 \rangle$ şi că listele de adiacenţă sunt

$$\begin{aligned} N[v_1] &= \langle s, v_2, v_3 \rangle , \\ N[v_2] &= \langle s, v_1, v_3, v_4 \rangle , \\ N[v_3] &= \langle v_1, v_2, v_4, t \rangle , \\ N[v_4] &= \langle v_2, v_3, t \rangle . \end{aligned}$$

27.5-2 ⋆ Dorim să implementăm un algoritm de preflux în care gestionăm o structură de tip coadă de vârfuri de depăşire. Algoritmul descarcă repetat vârful din capul cozii şi oricare vârfuri, care nu sunt de depăşire înainte de descărcare, dar sunt de depăşire după aceea, sunt plasate la sfârşitul cozii. După ce vârful din capul cozii este descărcat, el este eliminat. Când coada se goleşte, algoritmul se termină. Arătaţi că acest algoritm poate fi implementat să calculeze un flux maxim în timpul $O(V^3)$.

27.5-3 Arătaţi că algoritmul generic funcţionează şi dacă RIDICĂ actualizează $h[u]$ calculând simplu $h[u] \leftarrow h[u] + 1$. Cum afectează această schimbare analiza lui MUTĂ-ÎN-FAŢĂ?

27.5-4 ⋆ Demonstraţi că dacă descărcăm întotdeauna un cel mai înalt vârf de depăşire, metoda prefluxului poate fi făcută să se execute în timpul $O(V^3)$.

Probleme

27-1 Problema evadării

O ***reţea*** $n \times n$ este un graf neorientat format din n linii şi n coloane de vârfuri, conform figurii 27.12. Notăm vârful din al i-lea rând şi a j-a coloană prin (i, j). Toate vârfurile dintr-o reţea au exact patru vecini, cu excepţia vârfurilor marginale, care sunt punctele (i, j) pentru care $i = 1$, $i = n$, $j = 1$ sau $j = n$.

Fiind date $m \leq n^2$ puncte de plecare (x_1, y_1), (x_2, y_2),..., (x_m, y_m) pe reţea, ***problema evadării*** constă în a determina dacă există sau nu drumuri disjuncte compuse din câte m vârfuri din punctele de plecare spre oricare m puncte diferite de pe frontieră. De exemplu, reţeaua din figura 27.12(a) are o evadare, dar reţeaua din figura 27.12(b) nu are nici una.

a. Să considerăm o reţea de transport în care vârfurile, la fel ca şi muchiile, au asociate capacităţi. Adică fluxul pozitiv al reţelei care intră în orice vârf dat este constrâns la o capacitate. Arătaţi că determinarea fluxului maxim într-o reţea cu muchii şi vârfuri având asociate capacităţi poate fi redusă la o problemă obişnuită de flux maxim pe o reţea de transport de mărime comparabilă.

b. Descrieţi un algoritm eficient pentru rezolvarea problemei evadării şi analizaţi timpul său de execuţie.

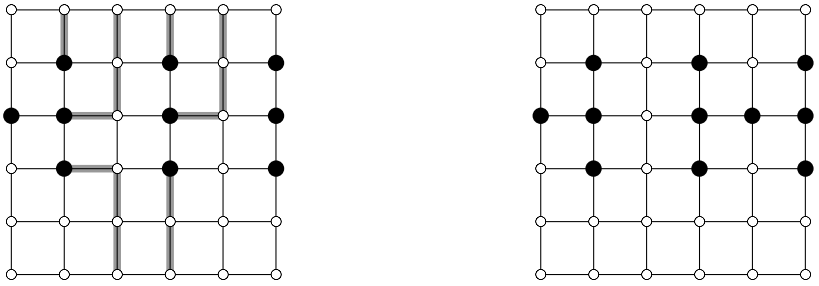


Figura 27.12 Reţele pentru problema evadării. Punctele de plecare sunt negre, iar celelalte vârfuri ale reţelei sunt albe. **(a)** O reţea cu o evadare, schiţată prin drumuri înnegrite. **(b)** O reţea fără nici o evadare.

27-2 Acoperire cu drumuri minime

O ***acoperire cu drumuri*** a unui graf orientat $G = (V, E)$ este o mulţime P de drumuri disjuncte astfel încât fiecare vârf din V este inclus în exact un drum din P. Drumurile pot începe şi se pot termina oriunde şi pot avea orice lungime, inclusiv 0. O ***acoperire cu drumuri minime*** a lui G este o acoperire de drumuri ce conţine cât mai puţine drumuri posibile.

a. Realizaţi un algoritm eficient pentru găsirea unei acoperiri cu drumuri minime a unui graf orientat aciclic $G = (V, E)$. (*Indicaţie:* Presupunând că $V = \{1, 2, \ldots, n\}$, construiţi un graf $G' = (V', E')$, pentru care

$$\begin{aligned} V' &= \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\}\,, \\ E' &= \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}\,, \end{aligned}$$

şi aplicaţi un algoritm de flux maxim.)

b. Funcţionează algoritmul creat şi pe grafuri orientate care conţin cicluri? Argumentaţi.

27-3 Experimentele navetei spaţiale

Profesorul Spock este consultant la NASA, unde se planifică o serie de zboruri ale navetei spaţiale şi trebuie să se decidă ce experimente comerciale să realizeze şi ce instrumente să aibă la bord fiecare zbor. Pentru fiecare zbor, NASA ia în considerare o mulţime $E = \{E_1, E_2, \ldots, E_m\}$ de experimente, iar sponsorul experimentului comercial E_j a fost de acord să plătească p_j dolari pentru rezultatele acestui experiment. Experimentele utilizează o mulţime $I = \{I_1, I_2, \ldots, I_n\}$ de instrumente; fiecare experiment E_j necesită toate instrumentele dintr-o submulţime $R_j \subseteq I$. Costul transportării instrumentului I_k este c_k dolari. Rolul profesorului este de a găsi un algoritm eficient care să determine ce experimente să se realizeze şi care instrumente să fie transportate de un anumit zbor, cu scopul de a mări venitul net la maximum, care este venitul total rezultat din experimente minus costul total al instrumentelor transportate.

Să considerăm următoarea reţea G. Reţeaua conţine un vârf sursă s, vârfurile $I_1, I_2, \ldots, I_n$, vârfurile $E_1, E_2, \ldots, E_m$ şi un vârf de destinaţie t. Pentru $k = 1, 2 \ldots, n$, există o muchie (s, I_k) de capacitate c_k, iar pentru $j = 1, 2, \ldots, m$ există o muchie (E_j, t) de capacitate p_j. Pentru

$k = 1, 2 \ldots, n$ şi $j = 1, 2, \ldots, m$, dacă $I_k \in R_j$, atunci există o muchie (I_k, E_j) de capacitate infinită.

a. Arătaţi că dacă $E_j \in T$ pentru o tăietură de capacitate finită (S, T) a lui G, atunci $I_k \in T$ pentru orice $I_k \in R_j$.

b. Arătaţi cum se determină câştigul net maxim a lui NASA pe baza capacităţii tăieturii minime a lui G şi valorilor date p_j.

c. Construiţi un algoritm eficient pentru a determina care experimente să fie realizate şi care instrumente să fie transportate. Analizaţi timpul de execuţie al algoritmului construit în termenii m, n şi $r = \sum_{j=1}^{m} |R_j|$.

27-4 Actualizarea fluxului maxim

Fie $G = (V, E)$ o reţea de transport cu sursa s, destinaţia t şi capacităţi numere întregi. Să presupunem că este dat un flux maxim în G.

a. Presupunem că se măreşte cu 1 capacitatea unei singure muchii $(u, v) \in E$. Realizaţi un algoritm cu timp de execuţie $O(V + E)$ pentru a actualiza fluxul maxim.

b. Presupunem că se micşorează cu 1 capacitatea unei singure muchii $(u, v) \in E$. Realizaţi un algoritm cu timp de execuţie $O(V + E)$ pentru a actualiza fluxul maxim.

27-5 Fluxul maxim prin scalare

Fie $G = (V, E)$ o reţea de transport cu sursa s, destinaţia t şi o capacitate număr intreg $c(u, v)$ pe fiecare muchie $(u, v) \in E$. Fie $C = \max_{(u,v)\in E} c(u, v)$.

a. Arătaţi că o taietură minimă a lui G are cel mult capacitatea $C|E|$.

b. Pentru un număr dat K, arătaţi că un drum de ameliorare de capacitate cel puţin egală cu K poate fi găsit în timp $O(E)$, dacă există un astfel de drum.

Următoarea modificare a algoritmului METODA-FORD-FULKERSON poate fi utilizată să calculeze un flux maxim în G.

FLUX-MAXIM-PRIN-SCALARE(G, s, t)
1: $C \leftarrow \max_{(u,v)\in E} c(u, v)$
2: iniţializează fluxul f cu 0
3: $K \leftarrow 2^{\lfloor \lg C \rfloor}$
4: **cât timp** $K \geq 1$ **execută**
5: **cât timp** există un drum de ameliorare p de capacitate cel puţin K **execută**
6: măreşte fluxul f de-a lungul lui p
7: $K \leftarrow K/2$
8: **returnează** f

c. Argumentaţi că FLUX-MAXIM-PRIN-SCALARE returnează fluxul maxim.

d. Arătaţi că valoarea capacităţii reziduale a unei tăieturi minime a grafului rezidual G_f este cel mult $2K|E|$ de fiecare dată când este executată linia 4.

e. Argumentaţi că ciclul interior **cât timp** din liniile 5–6 este executat de $O(E)$ ori pentru fiecare valoare a lui K.

f. Demonstraţi că FLUX-MAXIM-PRIN-SCALARE poate fi implementat să se execute în timpul $O\left(E^2 \lg C\right)$.

27-6 Fluxul maxim cu restricţii de capacitate superioară şi inferioară

Să presupunem că fiecare muchie (u, v) dintr-o reţea de transport $G = (V, E)$ nu are numai o margine superioară $c(u, v)$ pentru fluxul de la u la v, ci şi o margine inferioară $b(u, v)$. Adică orice flux f în reţea trebuie să satisfacă $b(u, v) \leq f(u, v) \leq c(u, v)$. Este posibil ca într-o astfel de reţea să nu existe nici un flux.

a. Demonstraţi că dacă f este un flux în G, atunci $|f| \leq c(S, T) - b(T, S)$ pentru orice tăietură (S, T) a lui G.

b. Demonstraţi că valoarea unui flux maxim în reţea, dacă există, este valoarea minimă a lui $c(S, T) - b(T, S)$ pe toate tăieturile (S, T) din reţea.

Fie $G = (V, E)$ o reţea de transport cu funcţiile de capacitate superioară şi inferioară c şi b şi fie s şi t sursa şi respectiv destinaţia lui G. Construiţi reţeaua obişnuită de flux $G' = (V', E')$ cu funcţia de capacitate superioară c', sursa s' şi destinaţia t' după cum urmează:

$$\begin{aligned} V' &= V \cup \{s', t'\}, \\ E' &= E \cup \{(s', v) : v \in V\} \cup \{(u, t') : u \in V\} \cup \{(s, t), (t, s)\}. \end{aligned}$$

Atribuim capacităţi muchiilor după cum urmează. Pentru fiecare muchie $(u, v) \in E$ atribuim $c'(u, v) = c(u, v) - b(u, v)$. Pentru fiecare vârf $u \in V$, atribuim $c'(s', u) = b(V, u)$ şi $c'(u, t') = b(u, V)$. De asemenea, vom avea şi $c'(s, t) = c'(t, s) = \infty$.

c. Demonstraţi că există un flux în G dacă şi numai dacă există un flux maxim în G', astfel încât toate muchiile către destinaţia t' sunt saturate.

d. Realizaţi un algoritm care să găsească un flux maxim într-o reţea cu capacitate superioară şi inferioară sau demonstraţi că nu există nici un flux. Analizaţi timpul de execuţie al algoritmului construit.

Note bibliografice

Referinţe bune pentru reţele de transport şi algoritmi relativ la reţele de transport sunt Even [65], Lawler [132], Papadimitriou şi Steiglitz [154] şi Tarjan [188]. Goldberg, Tardos şi Tarjan [83] realizează o bună analiză a algoritmilor pentru probleme cu reţele de flux.

Metoda Ford-Fulkerson se datorează lui Ford şi Fulkerson [71], care au construit multe probleme în domeniul reţelelor de transport, inclusiv problemele fluxului maxim şi a cuplajului bipartit. Multe din implementările mai vechi ale metodei Ford-Fulkerson au găsit drumul de ameliorare folosind căutarea în lăţime; Edmonds şi Karp [63] au demonstrat că această strategie

duce la un algoritm în timp polinomial. Karzanov [119] a dezvoltat ideea prefluxurilor. Metoda prefluxului se datorează lui Goldberg [82]. Cel mai rapid algoritm de preflux la ora actuală este realizat de Goldberg şi Tarjan [85], care au atins timpul de execuţie $O\left(VE \lg\left(V^2/E\right)\right)$. Cel mai bun algoritm în prezent pentru cuplajul bipartit maxim, descoperit de Hopcroft şi Karp [101], se execută în timpul $O\left(\sqrt{V}E\right)$.

VII Capitole speciale

Introducere

În această parte, sunt prezentate probleme speciale de algoritmi care extind şi completează materialul tratat până acum în carte. Astfel, unele capitole introduc modele noi de calcul, cum ar fi circuite combinatoriale şi calculatoare paralele, altele acoperă domenii speciale, cum ar fi geometria computaţională sau teoria numerelor. Ultimele două capitole pun în discuţie câteva dintre limitele cunoscute ale proiectării algoritmilor eficienţi şi introduc tehnici care tratează aceste limite.

În capitolul 28, se prezintă primul nostru model de calcul paralel: reţelele de comparare. Într-o formulare mai puţin precisă, se poate spune că o reţea de comparare este un algoritm care ne permite să efectuăm mai multe comparaţii simultan. În acest capitol, se arată cum se poate realiza o astfel de reţea de comparare, care sortează n numere în $O(\lg^2 n)$ unităţi de timp.

În capitolul 29, se introduce un alt model de calcul paralel: circuite combinatoriale. În acest capitol se arată cum se pot aduna două numere de câte n biţi în $O(\lg n)$ unităţi de timp, cu ajutorul unui circuit combinatorial. Deasemenea acelaşi model ne arată cum se înmulţesc două numere de câte n biţi în $O(\lg n)$ unităţi de timp.

În capitolul 30, se introduce un model general de calcul paralel, numit model PRAM. Capitolul prezintă tehnicile paralele de bază, incluzând saltul de pointeri, calculul prefixului şi tehnica turului Euler. Majoritatea tehnicilor sunt ilustrate pe structuri de date simple, incluzând liste şi arbori. Capitolul tratează, de asemenea, problemele generale ale calculului paralel, incluzând atât eficienţa lucrului cu memorie partajată cât şi a accesului concurent la acesta. Se demonstrează teorema lui Brent, care ne arată cum poate fi simulat, eficient, un circuit combinatorial de un calculator paralel. Capitolul se termină cu un algoritm aleator eficient pentru list ranking şi cu un algoritm deterministic cu o eficienţă remarcabilă pentru ruperea simetriei într-o listă.

În capitolul 31, se studiază eficienţa algoritmilor de calcul matriceal. Se începe cu algoritmul lui Strassen, care poate înmulţi două matrice de dimeniune $n \times n$ în $O(n^{2,81})$ unităţi de timp. Se prezintă apoi două metode generale – descompunerea LU şi descompunerea LUP – pentru rezolvarea sistemelor de ecuaţii liniare, prin metoda de eliminare a lui Gauss în $O(n^3)$ unităţi de timp. Se va arăta că metoda lui Strassen poate conduce la algoritmi mai rapizi, pentru rezolvarea sistemelor liniare, şi, de asemenea, că inversarea matricilor şi produsul matricilor se pot realiza la fel de repede asimptotic. La sfârşitul capitolului, se arată cum se poate obţine o soluţie aproximativă cu ajutorul metodei celor mai mici pătrate, atunci când sistemul de ecuaţii nu are soluţie exactă.

În capitolul 32, se studiază operaţiile cu polinoame, şi prezintă cum, o bine cunoscută metodă pentru prelucrarea semnalelor – transformata Fourier rapidă (TFR) – poate fi utilizată la înmulţirea a două polinoame de grad n în $O(n \lg n)$ unităţi de timp. Se studiază, de asemenea, implementările eficiente ale metodei TFR, inclusiv un circuit paralel.

În capitolul 33, se prezintă algoritmi din teoria numerelor. După o scurtă trecere în revistă a rezultatelor teoriei elementare a numerelor, se prezintă algoritmul lui Euclid pentru calculul celui mai mare divizor comun. Apoi, sunt prezentate metode de rezolvare a sistemelor de congruenţe liniare şi metode de ridicare la putere modulo n. Se prezintă, apoi, un rezultat interesant al teoriei numerelor: criptosistemul RSA cu cheie publică. Acest criptosistem nu numai că permite cifrarea textelor, dar este capabil să realizeze şi semnături digitale. Capitolul tratează, apoi, testul aleator de primalitate de tip Miller-Rabin, care poate fi utilizat pentru a obţine, într-un

mod eficient, numere prime mari – cerinţă esenţială în sistemul RSA. La sfârşitul capitolului, se prezintă algoritmul euristic "rho" al lui Pollard pentru factorizarea întregilor şi toată problematica factorizării întregilor.

În capitolul 34, se studiază problema găsirii tuturor aparţiiilor unei secvenţe de caractere într-un text, problemă frecventă în editarea textelor. Se începe cu o abordare elegantă datorată lui Rabin şi Carp. Apoi, după examinarea unei soluţii eficiente bazată pe automate finite, se trece la prezentarea algoritmului Knuth-Morris-Pratt, care găseşte soluţia eficient printr-o preprocesare inteligentă a secvenţei ale cărei aparţii se caută. Capitolul se încheie cu prezentarea algoritmului euristic Boyer-Moore pentru rezolvarea acestei probleme.

Tema capitolului 35 este geometria computaţională. După discutarea problemelor de bază ale geometriei computaţionale, se prezintă o metodă de "baleiere" cu ajutorul căreia se poate determina, eficient, dacă o mulţime de segmente de linii conţine sau nu, segmente care se intersectează. Puterea algoritmului de baleiere este ilustrată şi de doi algoritmi eficienţi – elaboraţi de Graham şi Jarvis – pentru găsirea înfăşurătorii convexe a unei mulţimi de puncte. La sfârşitul capitolului, se prezintă un algoritm eficient pentru găsirea perechii de puncte cele mai apropiate dintr-o mulţime de puncte în plan.

Capitolul 36 este dedicat problemelor NP-complete. Multe probleme computaţionale sunt NP-complete şi, deci, nu se cunosc algoritmi polinomiali pentru rezolvarea lor. Sunt prezentate tehnici pentru a afla dacă o problemă este NP-completă sau nu. Multe probleme clasice sunt NP-complete: determinarea dacă un graf conţine un ciclu hamiltonian, determinarea dacă o formulă booleană este satisfiabilă, şi determinarea dacă într-o mulţime de numere există o submulţime astfel încât suma elementelor submulţimii să fie egală cu o valoare dată. În acest capitol se demonstrează, de asemenea, că faimoasa problemă a comis-voiajorului este NP-completă.

În capitolul 37, se arată cum pot fi utilizaţi într-un mod eficient algoritmii de aproximare, pentru a găsi soluţii aproape optime. Pentru multe probleme NP-complete există soluţii aproximative (aproape optime), relativ uşor de obţinut. Pentru alte probleme, chiar cei mai buni algoritmi de aproximare cunoscuţi au randament din ce în ce mai scăzut o dată cu creşterea dimensiunilor datelor problemei. Sunt şi probleme la care, cu cât timpul de rezolvare creşte, obţinem soluţii aproximative din ce în mai bune. Capitolul de faţă ilustrează astfel de posibilităţi pentru problema mulţimii exterior stabile, a comis-voiajorului, a acoperirii cu mulţimi şi a submulţimilor de sumă dată.

28 Reţele de sortare

În partea a II-a a cărţii, am studiat algoritmii de sortare pentru calculatoare seriale (RAM adică "random access machine") care permit executarea, la un moment dat, numai a unei singure operaţii. În acest capitol, vom investiga algoritmii de sortare bazaţi pe modelul de calcul al reţelelor de comparare, în care se pot efectua simultan mai multe operaţii de comparare.

Reţelele de comparare diferă de calculatoarele RAM în două aspecte esenţiale. În primul rând, ele pot efectua numai comparaţii. Astfel, un algoritm cum ar fi sortarea prin numărare (vezi secţiunea 9.2) nu se poate realiza cu ajutorul reţelelor de comparare. În al doilea rând, spre deosebire de calculatoarele RAM, unde operaţiile sunt efectuate secvenţial – adică una dupa alta –, în reţele de comparare operaţiile pot fi efectuate simultan, adică în "paralel". Aşa cum vom vedea mai târziu, această caracteristică ne permite realizarea unor reţele de comparare care sortează n valori într-un timp subliniar.

În secţiunea 28.1, definim reţelele de comparare şi reţelele de sortare. Vom da, de asemenea, o definiţie naturală a "timpului de execuţie" în cazul reţelelor de comparare, în funcţie de adâncimea reţelei. În secţiunea 28.2, se demonstrează "principiul zero-unu", care ne va uşura mult analiza corectitudinii reţelelor de sortare.

Reţeaua eficientă de sortare pe care o vom proiecta, este, de fapt, o variantă paralelă a algoritmului de sortare prin interclasare, studiat în secţiunea 1.3.1. Construcţia reţelei se va realiza în trei paşi. Secţiunea 28.3 prezintă proiectarea aşa-numitului algoritm de sortare bitonic, care va fi celula de bază în construcţia reţelei de sortare. Modificăm acest algoritm în secţiunea 28.4 ca să poată interclasa două secvenţe ordonate într-o singură secvenţă ordonată. În final, în secţiunea 28.5, vom asambla aceste reţele de interclasare într-o reţea de sortare, care sortează n elemente în $O(\lg^2 n)$ unităţi de timp.

28.1. Reţele de comparare

Reţelele de sortare sunt reţele de comparare care, întotdeauna, îşi sortează datele se intrare, deci are sens să începem discuţia noastră cu reţelele de comparare şi caracteristicile lor. O reţea de comparare este compusă numai din fire şi comparatori. Un ***comparator***, cum este prezentat în figura 28.1(a), este un aparat cu două intrări, x şi y, şi două ieşiri, x' şi y', care realizează următoarea funcţie:

$$\begin{aligned} x' &= \min(x, y), \\ y' &= \max(x, y). \end{aligned}$$

În loc de reprezentarea unui comparator, dată în figura 28.1(a), vom folosi una mai simplă, adoptând convenţia ca, în loc de comparator, să desenăm, pur şi simplu, o bară verticală, ca în figura 28.1(b). Intrările se află în partea stângă, ieşirile în partea dreaptă cu precizarea că, la ieşire, valoarea mai mică se află în partea de sus, iar cea mai mare în partea de jos. Comparatorul poate fi privit ca un aparat care sortează cele două intrări ale sale.

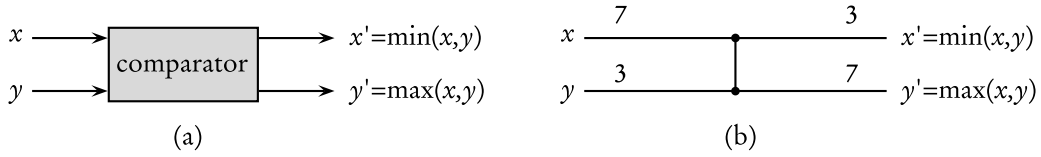


Figura 28.1 **(a)** Un comparator cu două intrări, x şi y, şi două ieşiri, x' şi y'. **(b)** Acelaşi comparator desenat ca o simplă bară verticală. Intrări: $x = 7, y = 3$, ieşiri: $x' = 3, y' = 7$.

Vom presupune că fiecare comparator operează în $O(1)$ unităţi de timp. Cu alte cuvinte, presupunem că timpul scurs de la apariţia intrărilor x şi y până la producerea ieşirilor x' şi y' este constant.

Un ***fir*** transmite informaţii dintr-un loc într-altul. Firele de ieşire ale unui comparator pot fi legate la intrările unui alt comparator, în rest, firele sunt sau fire de intrare în reţea, sau fire de ieşire din reţea. În tot acest capitol, presupunem că o reţea de comparare conţine n ***fire de intrare***, notate cu $a_1, a_2, \ldots, a_n$, prin care datele de sortat intră în reţea, şi n ***fire de ieşire***, notate cu $b_1, b_2, \ldots, b_n$, prin care sunt furnizate datele rezultat. De asemena se poate vorbi de ***secvenţa de intrare*** $\langle a_1, a_2, \ldots, a_n \rangle$ şi de ***secvenţa de ieşire*** $\langle b_1, b_2, \ldots, b_n \rangle$ care sunt valorile ataşate firelor de intrare, respectiv de ieşire. Deci nu facem deosebire între fire şi valorile firelor, ceea ce nu va conduce însă la nici o confuzie.

În figura 28.2 se poate vedea o ***reţea de comparare***, constă dintr-o mulţime de comparatori interconectaţi prin fire. Reprezentăm o reţea de comparare cu n intrări prin n ***linii*** orizontale, dintre care unele sunt unite prin bare verticale, reprezentând comparatorii. Să remarcăm că o linie *nu* este un singur fir, ci mai degrabă o secvenţă de fire distincte, care leagă diferiţi comparatori. De exemplu, linia de sus, în figura 28.2, reprezintă trei fire: firul de intrare a_1, care este legat la o intrare a comparatorului A; firul care leagă ieşirea de sus a comparatorului A cu o intrare a comparatorului C; şi, în fine, firul de ieşire b_1, care reprezintă ieşirea de sus a comparatorului C. Fiecare intrare a unui comparator este legată la un fir care, fie este unul din cele n fire de intrare $a_1, a_2, \ldots, a_n$ ale reţelei de comparare, fie este conectată la una dintre ieşirele unui alt comparator. În mod similar, fiecare ieşire a unui comparator este legată la un fir care, fie este unul din cele n fire de ieşire $b_1, b_2, \ldots, b_n$ ale reţelei, fie este conectată la una dintre intrările unui alt comparator. Cerinţa principală, la interconectarea comparatorilor, este ca graful interconexiunilor să fie aciclic: respectiv, dacă pornim de la ieşirea unui comparator şi ajungem la intrarea unui alt comparator, trecând prin alte ieşiri şi intrări, nu putem ajunge niciodată înapoi de unde am pornit şi nu putem trece niciodată prin acelaşi comparator de două ori. Astfel se poate desena o reţea de comparare, ca în figura 28.2, cu intrările la stânga şi ieşirele la dreapta, datele mişcându-se de la stânga la dreapta.

Fiecare comparator produce valoarea sa de ieşire în momentul în care are la dispoziţie ambele intrări. În figura 28.2(a), de exemplu, presupunem că, în momentul 0, apare la intrare secvenţa $\langle 9, 5, 2, 6 \rangle$. În momentul 0, numai comparatorii A şi B dispun de intrarea lor. Presupunând că fiecare comparator necesită o unitate de timp pentru producerea ieşirii, comparatorii A şi B produc ieşirile lor în momentul 1; rezultatul se poate vedea în figura 28.2(b)). Să observăm că ieşirile comparatorilor A şi B sunt produse "în paralel". În momentul 1, comparatorii C, D au la dispoziţie intrările lor, dar E încă nu. Un moment mai târziu, aceşti comparatori produc ieşirile lor, cum se vede în figura 28.2(c). Comparatorii C şi D lucrează, de asemenea, în paralel. Ieşirea de sus a comparatorului C şi cea de jos a comparatorului D sunt conectate la ieşirile b_1, respectiv,

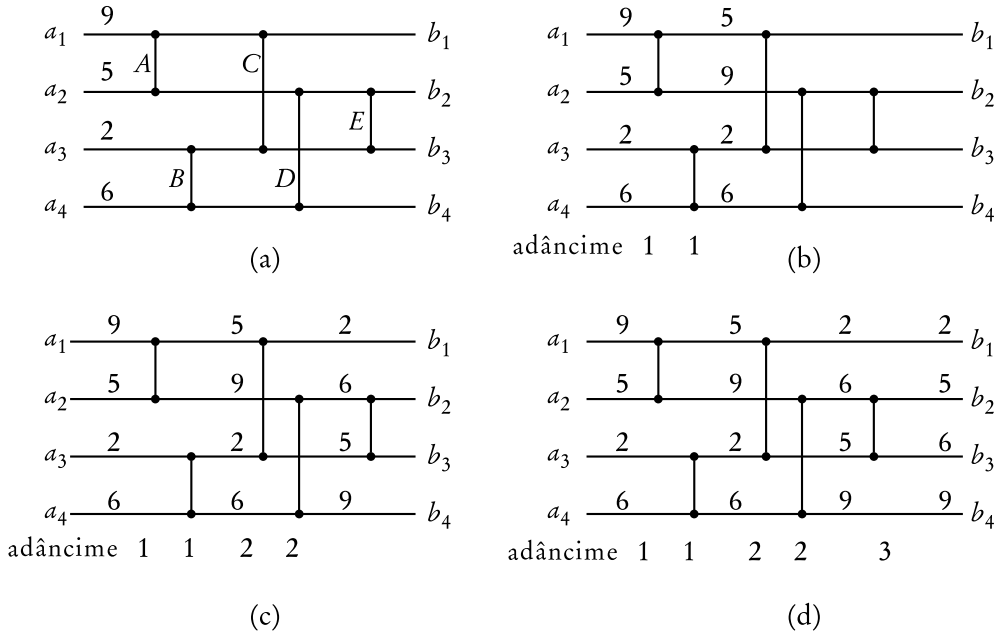


Figura 28.2 **(a)** O reţea de comparare cu 4 intrări şi 4 ieşiri, care este de fapt o reţea de sortare. La momentul 0, datele de intrare sunt scrise pe cele patru fire de intrare. **(b)** În momentul 1, datele scrise pe fire sunt ieşirile comparatorilor A şi B, care se găsesc la adâncimea 1. **(c)** În momentul 2, datele scrise pe fire sunt ieşirile comparatorilor C şi D, care sunt la adâncimea 2. Firele de ieşire b_1 şi b_4 au deja valorile lor finale, pe când firele b_2 şi b_3 încă nu. **(d)** În momentul 3, datele scrise pe fire sunt ieşirile comparatorului E de adâncime 3. Pe firele de ieşire b_2 şi b_3 se află deja valorile lor finale.

b_4 ale reţelei de comparare, şi, ca atare, acestea au valoarea lor finală în momentul 2. Între timp comparatorul E, având la dispoziţie intrările lui, produce în momentul 3 ieşirile corespunzătoare, cum se poate vedea în figura 28.2(d). Aceste valori vor fi ieşirile b_2 şi b_3 ale reţelei de comparare şi, astfel, se obţine secvenţa completă de ieşire $\langle 2, 5, 6, 9 \rangle$.

Presupunând că fiecare comparator, pentru a produce ieşirile sale, are nevoie de o unitate de timp, se poate defini "timpul de execuţie" al reţelei de comparare ca fiind timpul necesar producerii tuturor ieşirilor din momentul în care datele iniţiale sunt pe firele de intrare. Acest timp, este de fapt, numărul maxim de comparatori prin care o valoare de la un fir de intrare trebuie să treacă ca să ajungă la un fir de ieşire. Mai precis, se poate defini ***adâncimea*** unui fir după cum urmează. Un fir de intrare al unei reţele de comparare are adâncimea 0. Dacă un comparator are două fire de intrare cu adâncimea d_x, respectiv, d_y, atunci firele sale de ieşire au adâncimea $\max(d_x, d_y) + 1$. Pentru că într-o reţea de comparare nu există cicluri, adâncimea fiecărui fir este precis definită, şi vom defini adâncimea unui comparator ca fiind egală cu adâncimea firelor sale de ieşire. Figura 28.2 ilustrează adâncimea comparatorilor. Adâncimea unei reţele de comparare este adâncimea maximă a unui fir de ieşire sau, altfel spus, adâncimea maximă a unui comparator din reţea. De exemplu, reţeaua de comparare din figura 28.2 are adâncimea 3, deoarece comparatorul E are adâncimea 3. Dacă fiecare comparator are nevoie de o unitate de timp pentru producerea ieşirilor, şi intrările unei reţele de comparare apar în

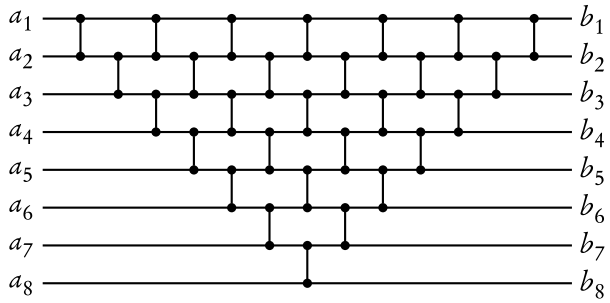


Figura 28.3 O reţea de sortare bazată pe metode de sortare prin inserare pentru a fi folosită în exerciţiul 28.1-6.

momentul 0, reţeaua de comparare cu adâncimea d are nevoie de d unităţi de timp pentru a produce toate ieşirile sale. Adâncimea unei reţele de comparare este, deci, egală cu timpul necesar producerii tuturor valorilor pe toate firele de ieşire.

O ***reţea de sortare*** este o reţea de comparare care produce, la ieşire, secvenţe monoton crescătoare (adică, $b_1 \leq b_2 \leq \ldots \leq b_n$) pentru *orice* secvenţă de intrare. Evident, nu orice reţea de comparare este şi reţea de sortare, dar reţeaua din figura 28.2 este. Pentru a demonstra de ce este aşa, să observăm că, după o unitate de timp, minimul celor patru valori va fi fie la ieşirea de sus a comparatorului A, fie la ieşirea de sus a comparatorului B. După 2 unităţi de timp, acest minim va fi la ieşirea de sus a comparatorului C. Un argument similar ne arată că maximul celor patru valori, după 2 unităţi de timp, ajunge la ieşirea de jos a comparatorului D. Comparatorul E asigură ca cele două valori de mijloc să ajungă în poziţia lor corectă, după momentul 3.

O reţea de comparare seamănă cu o procedură în care se specifică cum se fac comparările, dar nu seamănă cu o procedură în care dimensiunea ei (numărul comparatorilor) depinde de numărul intrărilor şi ieşirilor. De aceea vom descrie "familii" de reţele de comparare. În acest capitol vom dezvolta o familie SORTATOR, a reţelelor de sortare eficiente. În cadrul unei familii o anumită reţea va fi denumită după numele familiei şi numărul firelor de intrare (care ce este egal cu numărul firelor de ieşire). De exemplu, o reţea cu n intrări şi n ieşiri în famila SORTATOR se va numi SORTATOR$[n]$.

Exerciţii

28.1-1 Arătaţi ce valori apar pe toate firele reţelei din figura 28.2 dacă la intrare se dă secvenţa $\langle 9, 6, 5, 2 \rangle$.

28.1-2 Fie n o putere a lui 2. Arătaţi cum se poate construi o reţea de comparare cu n intrări şi n ieşiri de adâncime $\lg n$, în care ieşirea de sus să conţină totdeaunea valoarea minimă, iar ieşirea de jos totdeaunea valoarea maximă.

28.1-3 Profesorul Nielsen afirmă că, dacă inserăm un comparator oriunde într-o reţea de sortare, reţeaua obţinută va sorta corect orice secvenţă de intrare. Arătaţi că, inserând un comparator în reţeaua din figura 28.2, profesorul greşeşte, pentru că reţeaua obţinută nu mai sortează corect orice permutare a intrării.

28.1-4 Arătaţi că orice reţea de sortare cu n intrări are adâncimea cel puţin $\lg n$.

28.1-5 Arătaţi că numărul de comparări în orice reţea de sortare este $\Omega(n \lg n)$.

28.1-6 Demonstraţi că reţeaua de comparare din figura 28.3 este, de fapt, o reţea de sortare. Să se arate în ce măsură structura acestei reţele seamănă cu cea a algoritmului de sortare prin inserare (secţiunea 1.1).

28.1-7 O reţea de comparare cu n intrări şi c comparatori se poate reprezenta printr-o listă de c perechi de numere întregi de la 1 la n. Dacă două perechi au un număr în comun, ordinea comparatorilor respectivi în reţea se poate determina prin ordinea perechilor în listă. Pornind de la această reprezentare, descrieţi un algoritm (secvenţial) de complexitate $O(n + c)$ pentru determinarea adâncimii reţelei de comparare.

28.1-8 $\star$ Să presupunem că, pe lângă comparatorii standard, folosim şi comparatori "inversaţi", adică comparatori la care valoarea minimă apare pe firul de jos, şi cea maximă pe firul de sus. Arătaţi cum se poate converti o reţea de sortare care foloseşte c comparatori standard şi inversaţi într-o reţea de sortare care foloseşte numai c comparatori standard. Arătaţi că metoda de conversie găsită este corectă.

28.2. Principiul zero-unu

Principiul zero-unu exprimă faptul că, dacă o reţea de sortare funcţionează corect pentru intrări din mulţimea $\{0, 1\}$, atunci, funcţionează corect pentru orice numere de intrare. (Numerele de intrare pot fi numere întregi, reale, sau, în general, orice numere dintr-o mulţime ordonată liniar.) Principiul zero-unu ne permite ca, la construcţia reţelelor de sortare şi a altor reţele de comparare, să ne concentrăm atenţia numai asupra operaţiilor care se execută cu numerele 0 şi 1. După ce am construit o reţea care sortează corect orice secvenţă de 0 şi 1, pe baza principiului zero-unu, putem afirma că ea sortează, corect, orice secvenţă de intrare.

Demonstraţia principiului zero-unu se bazează pe noţiunea de funcţie monoton crescătoare (secţiunea 2.2).

Lema 28.1 Dacă o reţea de comparare transformă secvenţa de intrare $a = \langle a_1, a_2, \ldots, a_n \rangle$ în secvenţa de ieşire $b = \langle b_1, b_2, \ldots, b_n \rangle$, atunci, pentru orice funcţie f monoton crescătoare, reţeaua va transforma secvenţa de intrare $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ în secvenţa de ieşire $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle$.

Demonstraţie. La început vom demonstra că dacă f este o funcţie monoton crescătoare, atunci un singur comparator cu intrările $f(x)$ şi $f(y)$ ne va da ca rezultat $f(\min(x, y))$ şi $f(\max(x, y))$. După aceasta, lema se va demonstra folosind inducţia matematică.

Pentru a demonstra afirmaţia de mai sus, să considerăm un comparator cu intrările x şi y. Ieşirea de sus a comparatorului va produce $\min(x, y)$, iar cea de jos $\max(x, y)$. Să presupunem acum că aplicăm, la intrare, valorile $f(x)$ şi $f(y)$, cum se arată în figura 28.4. Se obţine astfel la ieşirea de sus a comparatorului valoarea $\min(f(x), f(y))$, iar la ieşirea de jos valoarea $\max(f(x), f(y))$. Deoarece funcţia f este monoton crescătoare, $x \leq y$ implică $f(x) \leq f(y)$. Prin urmare, obţinem următoarele:

$$\min(f(x), f(y)) = f(\min(x, y)), \quad \max(f(x), f(y)) = f(\max(x, y)).$$

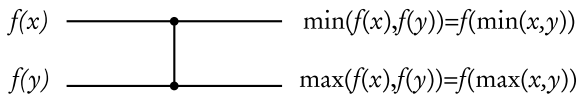


Figura 28.4 Operaţia de comparare folosită în demonstrarea lemei 28.1. Funcţia f este monoton crescătoare.

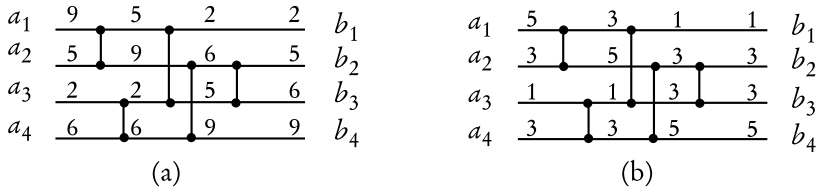


Figura 28.5 **(a)** Reţeaua de sortare din figura 28.2 cu secvenţa de intrare $\langle 9, 5, 2, 6\rangle$. **(b)** Aceeaşi reţea de sortare când se aplică la intrare funcţia monoton crescătoare $f(x) = f(\lceil x/2 \rceil)$. Valoarea pe fiecare fir este egală cu valoarea lui f aplicată pe valoarea firului corespunzător din reţeaua (a).

Deci, comparatorul produce valorile $f(\min(x, y))$ şi $f(\max(x, y))$ dacă intrările sunt $f(x)$ şi $f(y)$, şi afirmaţia este demonstrată.

Aplicând inducţia matematică pentru adâncimea unui fir oarecare dintr-o reţea de comparare, se poate demonstra o afirmaţie mai tare decât cea a lemei: dacă valoarea unui fir este a_i când secvenţa de intrare este a, atunci, valoarea ei va fi $f(a_i)$ când intrarea este secvenţa $f(a)$. Deoarece această afirmaţie este adevărată şi pentru firele de ieşire, demonstrând-o, demonstrăm şi lema.

La început, să considerăm un fir de adâncime 0, adică un fir de intrare a_i. Afirmaţia, în acest caz, este trivială: dacă $f(a)$ este secvenţa de intrare, atunci valoarea pe firul respectiv este $f(a_i)$. Să considerăm acum pentru pasul inductiv un fir de adâncime d ($d \geq 1$). Acest fir este ieşirea unui comparator de adâncime d, ale cărui fire de intrare sunt de adâncime strict mai mică de d. Din ipoteza inducţiei rezultă că dacă valorile de intrare ale comparatorului sunt a_i şi a_j când intrarea reţelei este secvenţa a, atunci valorile de intrare vor fi $f(a_i)$ şi $f(a_j)$ când secvenţa de intrare este $f(a)$. Conform afirmaţiei anterioare, pe firele de ieşire ale comparatorului apar valorile $f(\min(a_i, a_j))$ şi $f(\max(a_i, a_j))$. Deoarece atunci când secvenţa de intrare este a, pe aceste fire apar valorile $\min(a_i, a_j)$, respectiv $\max(a_i, a_j)$, lema este demonstrată. ∎

Figura 28.5 exemplifică aplicarea lemei. Ca reţea de sortare se foloseşte cea din figura 28.2, la care se aplică la intrare funcţia monoton crescătoare $f(x) = \lceil x/2 \rceil$. Valoarea pe fiecare fir este valoarea aplicării funcţiei f pe acelaşi fir în reţeaua din figura 28.2.

Dacă reţeaua de comparare este o reţea de sortare, lema 28.1 ne permite demonstrarea următorului rezultat remarcabil.

Teorema 28.2 (Principiul zero-unu) Dacă o reţea de comparare cu n intrări sortează corect toate cele 2^n secvenţe formate numai din 0-uri şi 1-uri, atunci va sorta corect orice secvenţă de n elemente.

Demonstraţie. Demontrăm prin reducere la absurd. Să presupunem că reţeaua sortează corect toate secvenţele formate numai din 0-uri şi 1-uri, dar există o secvenţă de valori arbitrare pe care nu o sortează corect. Adică, există o secvenţă de intrare $\langle a_1, a_2, \ldots, a_n\rangle$ astfel încât pentru

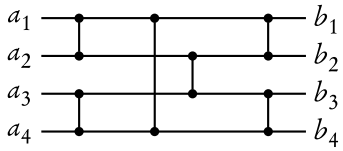


Figura 28.6 O reţea de sortare pentru sortarea a 4 numere.

valorile a_i şi a_j avem $a_i < a_j$, şi totuşi reţeaua îl plasează pe a_j înaintea lui a_i în secvenţa de ieşire. Să definim următoarea funcţie f monoton crescătoare:

$$f(x) = \begin{cases} 0, & \text{dacă } x \leq a_i, \\ 1, & \text{dacă } x > a_i. \end{cases}$$

Dacă în secvenţa de intrare $\langle a_1, a_2, \ldots, a_n \rangle$ elementul a_j este înaintea lui a_i, atunci, pe baza lemei 28.1, reţeaua plasează, în secvenţa de ieşire $\langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ pe $f(a_j)$ înaintea lui $f(a_i)$. Dar, deoarece $f(a_j) = 1$ şi $f(a_i) = 0$, obţinem contradicţia că reţeaua nu sortează corect secvenţa $\langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ formată numai din 0-uri şi 1-uri. ■

Exerciţii

28.2-1 Demonstraţi că, dacă se aplică o funcţie monoton crescătoare unei secvenţe sortate, se obţine tot o secvenţă sortată.

28.2-2 Demonstraţi că o reţea de comparare cu n intrări sortează corect secvenţa de intrare $\langle n, n-1, \ldots, 1 \rangle$ numai dacă sortează corect cele $n-1$ secvenţe $\langle 1, 0, 0, \ldots, 0, 0 \rangle$, $\langle 1, 1, 0, \ldots, 0, 0 \rangle$, $\ldots$, $\langle 1, 1, 1 \ldots, 1, 0 \rangle$.

28.2-3 Folosiţi principiul zero-unu pentru a demonstra că reţeaua de comparare din figura 28.6 este o reţea de sortare.

28.2-4 Enunţaţi şi demonstraţi un principiu analog principiului zero-unu pentru modelul arborelui de decizie. (*Indicaţie:* Atenţie la folosirea corectă a egalităţii.)

28.2-5 Demonstraţi că o reţea de sortare cu n intrări trebuie să conţină cel puţin un comparator între liniile i şi $(i+1)$ pentru orice $i = 1, 2, \ldots, n-1$.

28.3. O reţea de sortare a secvenţelor bitone

În construirea unei reţele eficiente de sortare, primul pas este realizarea unei reţele de comparare care sortează orice ***secvenţă bitonă***, secvenţă care creşte monoton, apoi descreşte monoton, sau poate fi redusă la o astfel de secvenţă prin permutări circulare. De exemplu, secvenţele $\langle 1, 4, 6, 8, 3, 2 \rangle$, $\langle 6, 9, 4, 2, 3, 5 \rangle$ şi $\langle 9, 8, 3, 2, 4, 6 \rangle$ sunt secvenţe bitone. Secvenţele bitone formate numai din 0-uri şi 1-uri au o structură foarte simplă. Ele au forma $0^i 1^j 0^k$ sau $1^i 0^j 1^k$ pentru anumite $i, j, k \geq 0$. De remarcat că secvenţele monoton crescătoare sau monoton descrescătoare sunt şi ele secvenţe bitone.

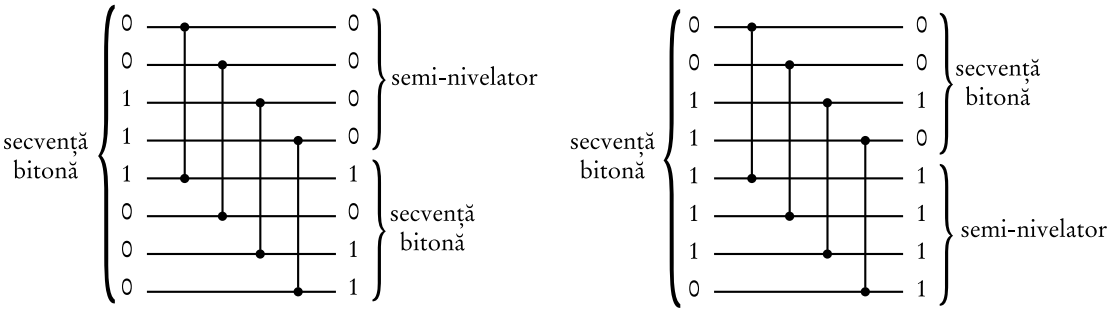


Figura 28.7 Reţeaua de comparare SEMI-NIVELATOR[8]. Se văd diferitele intrări şi ieşiri de 0-uri şi 1-uri. Un semi-nivelator asigură că în jumătatea de sus a ieşirii elementele nu sunt mai mari decât elementele din jumătatea de jos. Mai mult, ambele jumătăţi sunt bitone, şi cel puţin una din ele este constantă.

Un sortator bitonic, care va fi construit în continuare, este o reţea de comparare care sortează secvenţe bitone formate numai din 0-uri şi 1-uri. Exerciţiul 28.3-6 cere demonstrarea faptului că un sortator bitonic sortează corect secvenţe bitone de numere arbitrare.

Semi-nivelatorul

Sortatorul bitonic este compus din aşa-numite ***semi-nivelatoare***. Fiecare semi-nivelator este o reţea de comparare de adâncime 1, în care linia de intrare i se compară cu linia $i + n/2$ pentru $i = 1, 2, \ldots, n/2$. (Presupunem că n este par.) Figura 28.7 ne arată un semi-nivelator cu 8 intrări şi 8 ieşiri: SEMI-NIVELATOR[8].

Când intrarea unui semi-nivelator este o secvenţă bitonă formată numai din 0-uri şi 1-uri, semi-nivelatorul produce o secvenţă în care valorile mai mici sunt în jumătatea de sus, iar valorile mai mari în jumătatea de jos. De fapt, cel puţin una din cele două jumătăţi este ***constantă***, adică conţine numai sau 0-uri, sau numai 1-uri, de aici şi numele de "semi-nivelator". (De remarcat că orice secvenţă constantă este bitonă.) Următoarea lemă demonstrează această proprietate a semi-nivelatoarelor.

Lema 28.3 Dacă intrarea unui semi-nivelator este o secvenţă bitonă formată numai din 0-uri şi 1-uri, atunci secvenţa de ieşire are următoarele proprietăţi: ambele jumătăţi (atât cea de sus cât şi cea de jos) sunt bitone, fiecare element din jumătatea de sus este mai mic sau egal cu fiecare element din jumătatea de jos, cel puţin una din cele două jumătăţi este constantă.

Demonstraţie. Reţeaua de comparare SEMI-NIVELATOR[n] compară intrările i şi $i+n/2$ pentru $i = 1, 2, \ldots, n/2$. Fără a pierde din generalitate, se poate presupune că intrarea are forma $00\ldots011\ldots100\ldots0$. (Cazul când intrarea este de forma $11\ldots100\ldots011\ldots1$ este simetric.) Există trei cazuri posibile depinzând de blocul de 0-uri sau 1-uri consecutive în care se află elementul de mijloc $n/2$. Cazul în care acest element de mijloc se află în blocul de 1-uri, se împarte în două subcazuri. Cele patru cazuri sunt arătate în figura 28.8. În toate cazurile lema este adevărată. ∎

Sortatorul bitonic

Combinând recursiv semi-nivelatoare ca în figura 28.9, se poate construi un ***sortator bitonic***, care este o reţea care poate sorta secvenţe bitone. Prima parte a unui SORTATOR-BITONIC$[n]$ este un SEMI-NIVELATOR$[n]$, care, după lema 28.3, produce două secvenţe bitone având dimensiunea jumătate din cea iniţială, astfel încât fiecare element din jumătatea de sus este mai mic sau egal cu orice element din jumătatea de jos. Astfel, se poate completa sortarea folosind două copii ale lui SORTATOR-BITONIC$[n/2]$ pentru a sorta recursiv cele două jumătăţi. În figura 28.9(a) se poate vedea, explicit, recurenţa, iar în figura 28.9(b) sunt evidenţiate apelurile recursive consecutive, unde se pot vedea semi-nivelatoarele din ce în ce mai mici, care rezolvă sortarea. Adâncimea $D(n)$ a reţelei SORTATOR-BITONIC$[n]$ este dată de formula recursivă:

$$D(n) = \begin{cases} 0 & \text{dacă } n = 1, \\ D(n/2) + 1 & \text{dacă } n = 2^k \text{ şi } k \geq 1, \end{cases}$$

a cărei soluţii este $D(n) = \lg n$.

Astfel o secvenţă bitonă de elemente 0 şi 1 se poate sorta cu ajutorul unei reţele SORTATOR-BITONIC care are adâncimea $\lg n$. Pe baza unui principiu analog cu principiul zero-unu prezentat în exerciţiul 28.3-6, rezultă că această reţea poate sorta secvenţe bitone de numere arbitrare.

Exerciţii

28.3-1 Câte secvenţe bitone de lungime n formate numai din 0-uri şi 1-uri există?

28.3-2 Demonstraţi că reţeaua SORTATOR-BITONIC$[n]$, unde n este o putere a lui 2, conţine $\Theta(n \lg n)$ comparatori.

28.3-3 Descrieţi cum se poate construi un sortator bitonic de adâncime $O(\lg n)$, când n nu este o putere a lui 2.

28.3-4 Dacă intrarea unui semi-nivelator este o secvenţă bitonă de numere arbitrare, demonstraţi că ieşirea satisface următoarele proprietăţi: atât jumătatea de sus, cât şi cea de jos sunt bitone şi fiecare element din jumătatea de sus este mai mic sau egal cu orice element din jumătatea de jos.

28.3-5 Se consideră două secvenţe de 0-uri şi 1-uri. Demonstraţi că, dacă fiecare element dintr-o secvenţă este mai mic sau egal cu orice element din cealaltă secvenţă, atunci una din cele două secvenţe este constantă.

28.3-6 Demonstraţi următorul principiu pentru reţelele de sortare bitonă analog principiului zero-unu: o reţea de comparare, care sortează corect orice secvenţă bitonă de elemente 0 şi 1, sortează corect orice secvenţă bitonă de numere arbitrare.

28.4. Reţeaua de interclasare

Reţeaua noastră de sortare va fi realizată din ***reţele de interclasare*** care sunt reţele capabile să interclaseze două secvenţe de intrare sortate într-una de ieşire sortată. Modificăm reţeaua

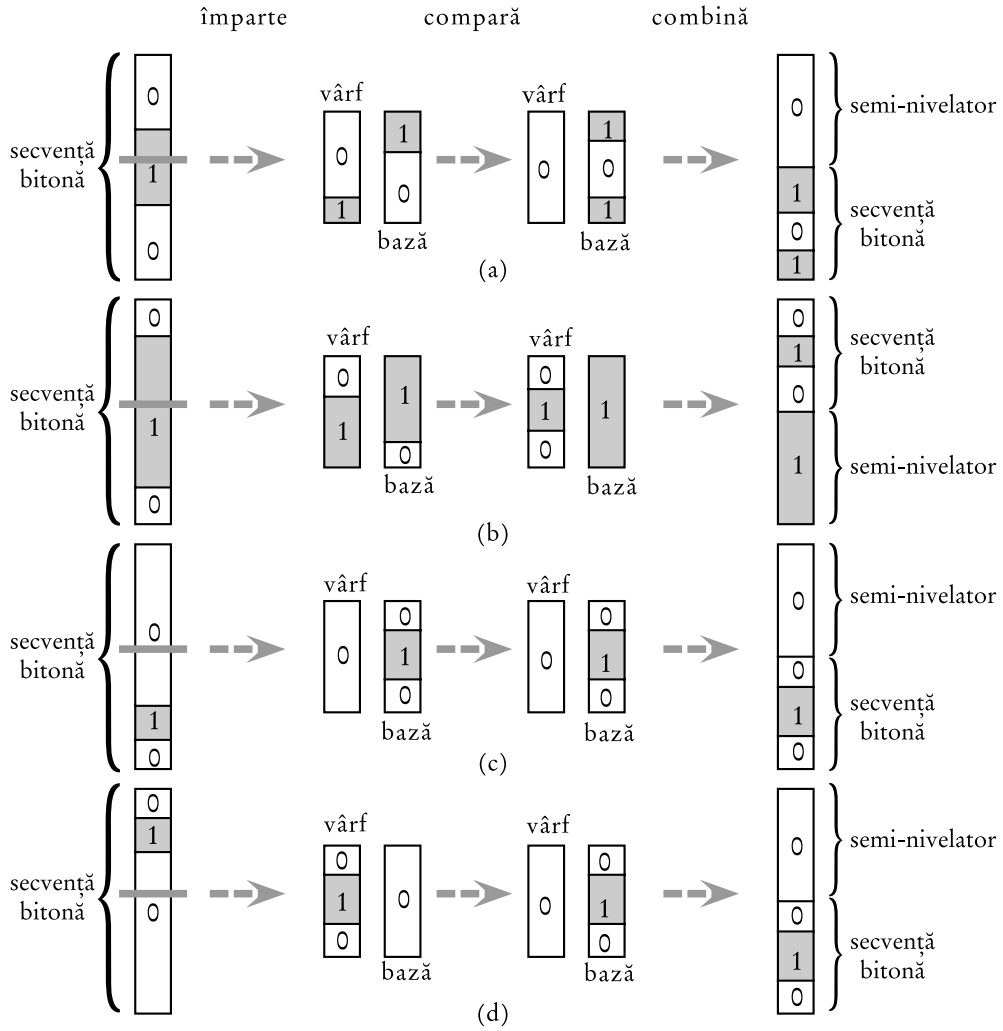


Figura 28.8 Comparările posibile în reţeaua SEMI-NIVELATOR$[n]$. Presupunem că secvenţa de intrare este bitonă cu elemente 0 şi 1, şi fără a pierde din generalitate presupunem că această secvenţă este de forma $00\ldots011\ldots100\ldots0$. Subsecvenţele formate numai din 0-uri sunt desenate în alb, iar cele formate numai din 1-uri în gri. Cele n elemente sunt divizate în două şi elementele i şi $i + n/2$ sunt comparate ($i = 1, 2, \ldots, n/2$). **(a)**–**(b)** Cazurile în care divizarea este la mijlocul secvenţei de 1-uri. **(c)**–**(d)** Cazurile în care divizarea este la mijlocul secvenţei de 0-uri. În toate cazurile, fiecare element din jumătatea de sus este mai mic sau egal cu orice element din jumătatea de jos. Ambele jumătăţi sunt bitone şi cel puţin una din cele două jumătăţi este constantă.

SORTATOR-BITONIC$[n]$ pentru a obţine o reţea INTERCLASOR$[n]$. Ca şi în cazul sortatorului bitonic, vom demostra corectitudinea reţelei de interclasare pentru secvenţe de intrare formate numai din 0-uri şi 1-uri. În exerciţiul 28.4-1 se cere extinderea demonstraţiei pentru secvenţe

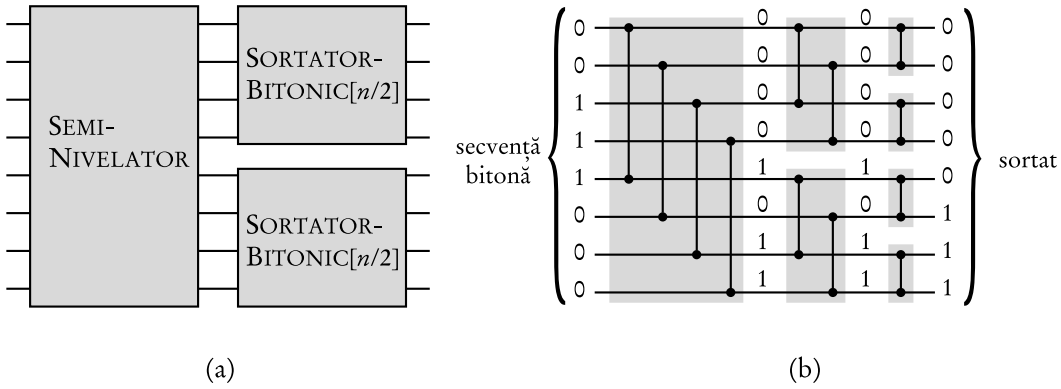


Figura 28.9 Reţeaua de comparare SORTATOR-BITONIC[n] pentru $n = 8$. **(a)** Construcţia recursivă: SEMI-NIVELATOR[n] urmat de două copii ale lui SORTATOR-BITONIC[$n/2$] care lucrează în paralel. **(b)** Reţeaua după desfăşurarea apelurilor recursive. Semi-nivelatoarele sunt haşurate. Pe fire sunt trecute ca exemple elemente 0 şi 1.

arbitrare.

Reţeaua de interclasare se bazează pe următoarea observaţie. Dacă avem două secvenţe sortate, inversăm ordinea elementelor din cea de a doua şi le concatenăm, obţinând astfel o secvenţă bitonă. De exemplu, fiind date secvenţele sortate $X = 00000111$ şi $Y = 00001111$, după inversarea lui Y, obţinem $Y^R = 11110000$. Concatenând X cu Y^R se obţine secvenţa 0000011111110000, care evident este o secvenţă bitonă. Deci, pentru a interclasa două secvenţe de intrare X şi Y, este suficientă o sortare bitonă pe secvenţa concatenată X cu Y^R.

Se poate realiza reţeaua INTERCLASOR[n] prin modificarea primului semi-nivelator din SORTATOR-BITONIC[n]. Esenţial este de a inversa, implicit, a doua jumătate a intrării. Fiind date două secvenţe de intrare sortate $\langle a_1, a_2, \ldots, a_{n/2}\rangle$ şi $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n\rangle$ care trebuie interclasate, ne interesează efectul sortării bitone a secvenţei $\langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1}\rangle$. Deoarece semi-nivelatorul reţelei SORTATOR-BITONIC[n] compară intrările i şi $(n/2+i)$ pentru $i = 1, 2, \ldots, n/2$, proiectăm prima etapă a reţelei de interclasare astfel încât să compare intrările i şi $n-i+1$. Figura 28.10 ilustrează corespondenţa. Singura subtilitate este că elementele de ieşire în partea de jos a primei etape a reţelei SORTATOR[n] sunt comparate în ordine inversă faţă de semi-nivelatorul iniţial. Deoarece inversa unei secvenţe bitone este tot bitonă, atât partea de sus cât şi cea de jos a ieşirii primei etape a reţelei de interclasare satisfac condiţiile lemei 28.3, şi, astfel, atât partea de sus cât şi partea de jos pot fi sortate în paralel pentru a produce ieşirea sortată a reţelei de interclasare.

Reţeaua de interclasare rezultată este ilustrată în figura 28.11. Numai prima etapă a reţelei INTERCLASOR[n] diferă de SORTATOR-BITONIC[n]. În consecinţă, adâncimea reţelei INTERCLASOR[n] este $\lg n$, adică aceeaşi ca la SORTATOR-BITONIC[n].

Exerciţii

28.4-1 Demonstraţi un principiu anolog principiului zero-unu pentru reţele de interclasare. Mai precis, arătaţi că, dacă o reţea de comparare poate interclasa două secvenţe monoton crescătoare formate numai din 0-uri şi 1-uri, atunci poate interclasa orice alte două secvenţe monoton

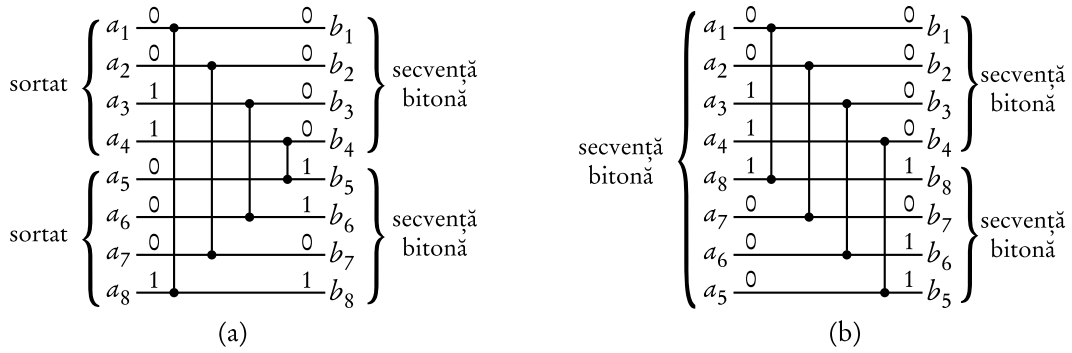


Figura 28.10 Compararea primei etape a reţelei INTERCLASOR$[n]$ cu cea a reţelei SEMI-NIVELATOR$[n]$ pentru $n = 8$. **(a)** Prima etapă a lui INTERCLASOR$[n]$ transformă cele două secvenţe monotone de intrare $\langle a_1, a_2, \ldots, a_{n/2}\rangle$ şi $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n\rangle$ în secvenţe bitone $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ şi $\langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n\rangle$. **(b)** Operaţiile echivalente ale reţelei SEMI-NIVELATOR$[n]$. Secvenţa bitonă de intrare $\langle a_1, a_2, \ldots, a_{n/2-1}, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+2}, a_{n/2+1}\rangle$ este transformată în două secvenţe bitone $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ şi $\langle b_n, b_{n-1}, \ldots, b_{n/2+1}\rangle$.

crescătoare de numere arbitrare.

28.4-2 Câte secvenţe de intrare diferite, formate numai din 0-uri şi 1-uri, trebuie aplicate la intrarea unei reţele de comparare pentru a verifica dacă este o reţea de interclasare?

28.4-3 Demonstraţi că o reţea oarecare, care poate interclasa un element cu $n - 1$ elemente sortate, pentru a produce o secvenţă sortată cu n elemente, trebuie să aibă adâncimea cel puţin $\lg n$.

28.4-4 $\star$ Se consideră o reţea de interclasare cu intrările $a_1, a_2, \ldots, a_n$, unde n este o putere a lui 2 şi în care cele două secvenţe, care trebuie sortate, sunt $\langle a_1, a_3, \ldots, a_{n-1}\rangle$ şi $\langle a_2, a_4, \ldots, a_n\rangle$. Demonstraţi că în astfel de reţele de interclasare numărul comparatorilor este $\Omega(n \lg n)$. De ce este interesantă această margine inferioară? (*Indicaţie:* Să se partiţioneze comparatorii în trei mulţimi.)

28.4-5 $\star$ Demonstraţi că orice reţea de interclasare, indiferent de ordinea intrărilor, necesită $\Omega(n \lg n)$ comparatori.

28.5. Reţeaua de sortare

Avem acum la dispoziţie toate elementele necesare realizării unei reţele care poate sorta orice secvenţă de intrare. Reţeaua de sortare SORTATOR$[n]$ foloseşte reţeaua de interclasare pentru a implementa o versiune paralelă a algoritumului de sortare prin interclasare din secţiunea 1.3.1. Construirea şi modul de funcţionare ale reţelei de sortare sunt ilustrate în figura 28.12.

Figura 28.12(a) ilustrează construirea recursivă a reţelei SORTATOR$[n]$. Cele n elemente de intrare sunt sortate recursiv de două copii ale reţelei SORTATOR$[n/2]$, care sortează fiecare

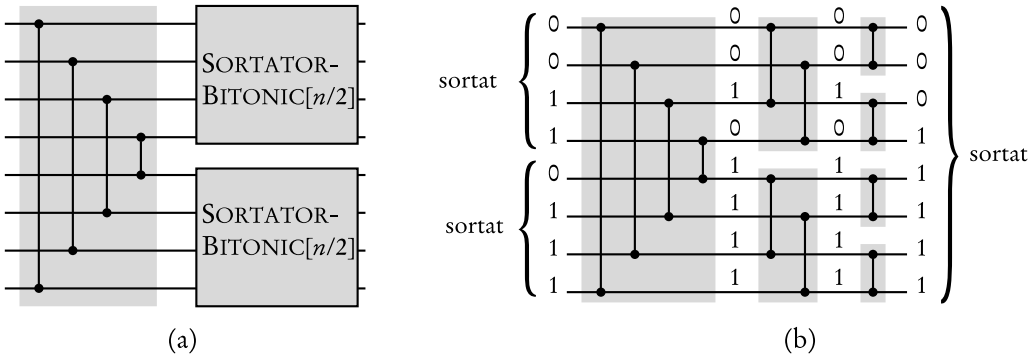


Figura 28.11 O reţea care interclasează două secvenţe de intrare sortate într-o singură secvenţă de ieşire sortată. Reţeaua INTERCLASOR[n] poate fi privită ca o reţea SORTATOR-BITONIC[n] în care primul semi-nivelator este modificat astfel încât să compare intrările i şi $n-i+1$ pentru $i = 1, 2, \ldots, n/2$. Aici $n = 8$. **(a)** Prima parte a reţelei este urmată de două copii paralele ale reţelei SORTATOR-BITONIC[$n/2$]. **(b)** Aceeaşi reţea după desfăşurarea apelului recursiv. Pe fire sunt trecute exemple de valori zero-unu, iar etapele sunt haşurate.

$n/2$ elemente în paralel. Cele două secvenţe sunt, pe urmă, interclasate de INTERCLASOR[n]. Condiţia de oprire este $n = 1$, când se poate folosi un singur fir pentru a sorta o secvenţă de 1 element, deoarece această secvenţă este deja sortată. Figura 28.12(b) ilustrează rezultatul desfăşurării apelului recursiv, iar figura 28.12(c) ilustrează reţeaua obţinută prin înlocuirea modulelor INTERCLASOR din figura 28.12(b) cu reţelele de interclasare corespunzătoare.

Datele în reţeaua SORTATOR[n] trec prin $\lg n$ etape. Fiecare intrare individuală în reţea este o secvenţă de lungime 1 deja sortată. Prima etapă a reţelei SORTATOR[n] este formată din $n/2$ copii de reţele INTERCLASOR[2], care sortează în paralel secvenţe de lungime 1 pentru a obţine secvenţe de lungime 2. A doua etapă este formată din $n/4$ copii ale reţelei INTERCLASOR[4], care interclasează perechi de secvenţe de lungime 2 pentru a obţine secvenţe de lungime 4. În general, pentru $k = 1, 2, \ldots, \lg n$, etapa k este formată din $n/2^k$ copii ale reţelei INTERCLASOR[2^k] care interclasează perechi de secvenţe de lungime 2^{k-1} pentru a obţine secvenţe de lungime 2^k. În etapa finală, se obţine o singură secvenţă sortată din toate elementele de intrare. Se poate demonstra prin inducţie că această reţea de sortare sortează secvenţele zero-unu, şi, astfel, pe baza principiului zero-unu (teorema 28.2), sortează secvenţe de valori arbitrare.

Adâncimea unei reţele de sortare poate fi studiată recursiv. Adâncimea $D(n)$ a reţelei SORTATOR[n] este adâncimea $D(n/2)$ a reţelei SORTATOR[$n/2$] (există două copii ale lui SORTATOR[$n/2$], dar ele lucrează în paralel) plus adâncimea $\lg n$ a reţelei INTERCLASOR[n]. Deci, adâncimea reţelei SORTATOR[n] este dată de formula de recurenţă:

$$D(n) = \begin{cases} 0 & \text{dacă } n = 1, \\ D(n/2) + \lg n & \text{dacă } n = 2^k \text{ şi } k \geq 1, \end{cases}$$

a cărei soluţie este $D(n) = \Theta(\lg^2 n)$. Deci, n numere pot fi sortate în paralel în $O(\lg^2 n)$ unităţi de timp.

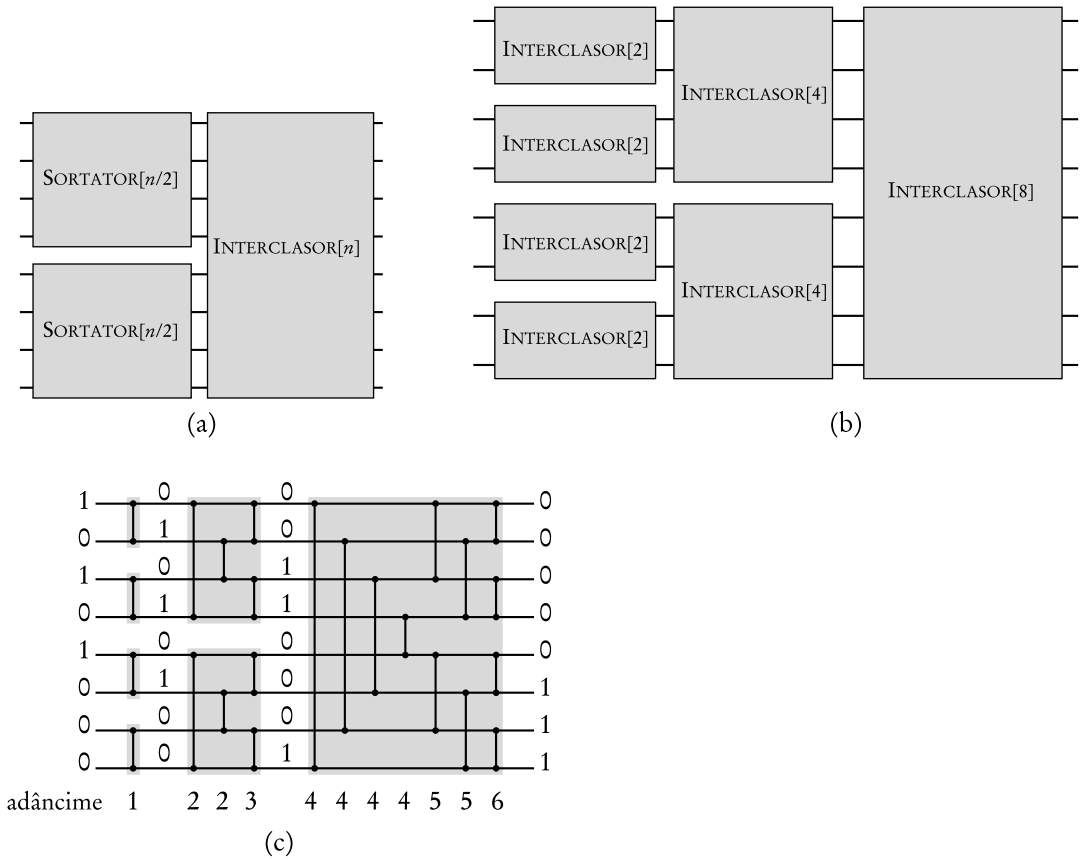


Figura 28.12 Reţeaua de sortare SORTATOR[n] construită prin combinarea recursivă a reţelelor de interclasare. **(a)** Construirea recursivă. **(b)** Desfăşurarea recursivă. **(c)** Înlocuirea modulelor INTERCLASOR cu reţelele de interclasare corespunzătoare. S-a indicat adâncimea fiecărui comparator şi s-au folosit exemple de valori 0 şi 1 pe fire.

Exerciţii

28.5-1 Câţi comparatori există în SORTATOR[n]?

28.5-2 Demonstraţi că adâncimea reţelei SORTATOR[n] este exact $(\lg n)(\lg n + 1)/2$.

28.5-3 Presupunem că modificăm un comparator astfel încât să primească la intrare două secvenţe sortate, de câte k elemente fiecare, şi să furnizeze la ieşirea "max" cele mai mari k elemente, şi la ieşirea "min" cele mai mici k elemente. Arătaţi că o reţea de sortare cu n intrări, care are astfel de comparatori modificaţi, poate sorta nk numere, în ipoteza că fiecare intrare a reţelei este o listă sortată de lungime k.

28.5-4 Se dau $2n$ elemente $\langle a_1, a_2, \ldots, a_{2n}\rangle$, pe care dorim să le partiţionăm în n cele mai mari şi n cele mai mici elemente. Arătaţi că acest lucru se poate face în timp constant după ce am sortat separat secvenţele $\langle a_1, a_2, \ldots, a_n\rangle$ şi $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n}\rangle$.

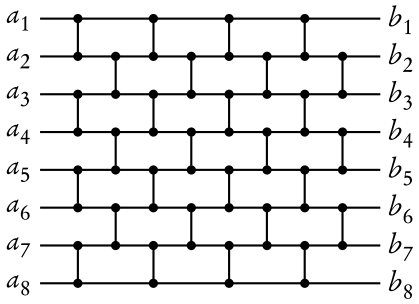


Figura 28.13 O reţea de sortare pară-impară cu 8 intrări.

28.5-5 ⋆ Fie $S(k)$ adâncimea unei reţele de sortare cu k intrări, iar $M(k)$ adâncimea unei reţele de interclasare cu $2k$ intrări. Să presupunem că avem de sortat o secvenţă de n numere şi ştim că fiecare element este într-o poziţie care diferă în cel mult k poziţii de poziţia lui corectă în secvenţa sortată. Arătaţi că se pot sorta n numere cu adâncimea $S(k) + 2M(k)$.

28.5-6 ⋆ Elementele unei matrice de $m \times m$ pot fi ordonate prin repetarea de k ori a următoarei proceduri:
1. Se sortează monoton crescător elementele fiecărei linii impare.
2. Se sortează monoton descrescător elementele fiecărei linii pare.
3. Se sortează monoton crescător elementele fiecărei coloane.
Cât este numărul de iteraţii k pentru ca procedura de mai sus să sorteze elementele matricei şi cum arată matricea?

Probleme

28-1 Reţele de sortare de transpoziţii
O reţea de comparare este o ***reţea de transpoziţii*** dacă fiecare comparator conectează linii adiacente, ca în reţeaua din figura 28.3.

a. Demonstraţi că o reţea de transpoziţii cu n intrări care sortează are $\Omega(n^2)$ comparatori.

b. Demonstraţi că o reţea de transpoziţii cu n intrări este o reţea de sortare dacă, şi numai dacă, sortează secvenţa $\langle n, n-1, \ldots, 1\rangle$. (*Indicaţie:* Să se folosească un argument asemănător celui din lema 28.1.)

O ***reţea de sortare pară-impară*** cu n intrări $\langle a_1, a_2, \ldots, a_n\rangle$ are n niveluri de comparatori. Figura 28.13 ilustrează o reţea de transpoziţii pară-impară cu 8 intrări. Conform figurii, pentru $i = 1, 2, \ldots, n$ şi $d = 1, 2, \ldots, n$ linia i este conectată la linia $j = i + (-1)^{i+d}$ printr-un comparator cu adâncimea d, dacă $1 \le j \le n$.

c. Demonstraţi că familia reţelelor de sortare pară-impară este de fapt o familie de reţele de sortare.

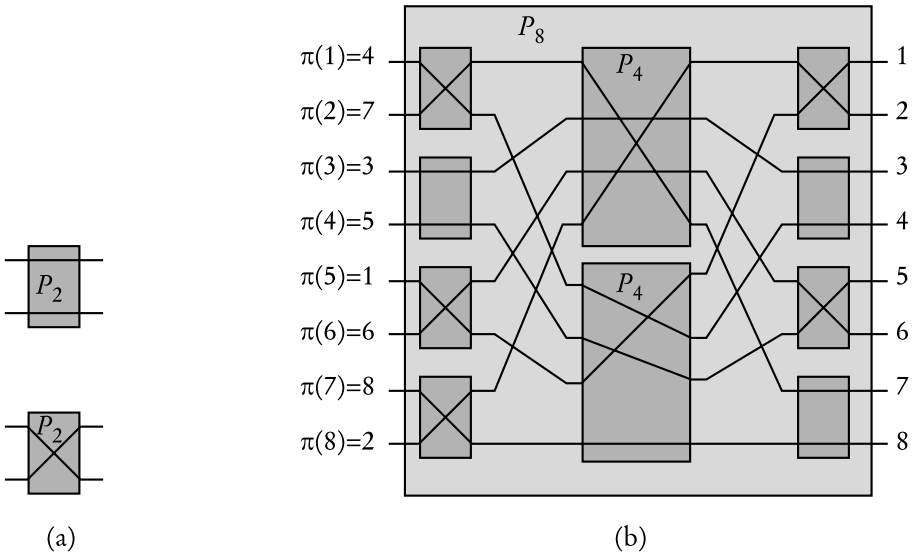


Figura 28.14 Reţea de permutări. **(a)** Reţeaua de permutări P_2 compusă dintr-un singur comutator, care are două poziţii, ca în figură. **(b)** Construcţia recursivă a reţelei P_8 compusă din 8 comutatori şi două reţele P_4. Comutatorii reţelei P_4 sunt setaţi pentru a realiza permutarea $\pi = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$.

28-2 Reţeaua de interclasare pară-impară a lui Batcher

În secţiunea 28.4 am văzut cum se poate construi o reţea de interclasare pornind de la sortare bitonă. În această problemă vom construi o ***reţea de interclasare pară-impară***. Presupunem că n este o putere a lui 2, şi dorim să interclasăm secvenţele sortate de pe liniile $\langle a_1, a_2, \ldots, a_n \rangle$ cu cele de pe liniile $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n} \rangle$. Construim recursiv două reţele de interclasare pară-impară care interclasează subsecvenţele sortate în paralel. Prima reţea interclasează secvenţa de pe liniile $\langle a_1, a_3, \ldots, a_{n-1} \rangle$ cu secvenţa de pe liniile $\langle a_{n+1}, a_{n+3}, \ldots, a_{2n-1} \rangle$ (elementele impare). A doua reţea interclasează elementele $\langle a_2, a_4, \ldots, a_n \rangle$ cu elementele $\langle a_{n+2}, a_{n+4}, \ldots, a_{2n} \rangle$ (elemente parele). Pentru a interclasa cele două secvenţe sortate punem câte un comparator între a_{2i-1} şi a_{2i} pentru $i = 1, 2, \ldots, n$.

a. Desenaţi o reţea de interclasare cu $2n$ intrări pentru $n = 4$.

b. Folosiţi principiul zero-unu pentru a demonstra că o reţea de interclasare pară-impară cu $2n$ intrări este de fapt o reţea de sortare.

c. Cât este adâncimea unei reţele de interclasare pară-impară cu $2n$ intrări?

28-3 Reţele de permutări

O ***reţea de permutări*** cu n intrări şi n ieşiri are comutatori care permit conectarea intrărilor cu ieşirile după toate permutările lui $n!$. Figura 28.14(a) arată o reţea de permutări P_2 cu 2 intrări 2 ieşiri care are un singur comutator care permite ca intrările să fie legate cu ieşirile direct sau în cruce.

a. Arătaţi că, dacă înlocuim, într-o reţea de sortare, fiecare comparator cu câte un comutator de tipul celui din figura 28.14(a), atunci reţeaua devine o reţea de permutare. Adică, pentru orice permutare π există un mod de a lega intrarea i cu ieşirea $\pi(i)$.

Figura 28.14(b) ilustrează construcţia recursivă a unei reţele de permutări P_8 cu 8 intrări şi 8 ieşiri, care foloseşte două copii ale lui P_4 şi 8 comutatori. Comutatorii sunt setaţi pentru a realiza permutarea $\pi = \langle \pi(1), \pi(2), \ldots, \pi(8) \rangle = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$ şi vor solicita (recursiv) ca partea de sus a lui P_4 să realizeze $\langle 4, 2, 3, 1 \rangle$, iar partea de jos $\langle 2, 3, 1, 4 \rangle$.

b. Arătaţi cum se poate realiza permutarea $\langle 5, 3, 4, 6, 1, 8, 2, 7 \rangle$ cu P_8, trasând poziţionarea comutatorilor şi permutările realizate de cele două P_4. Fie n o putere a lui 2. Să se definească recursiv P_n cu ajutorul a două reţele $P_{n/2}$, asemănător cazului P_8.

c. Scrieţi un algoritm (RAM obişnuit), de complexitate $O(n)$, care poziţionează cei n comutatori care sunt legaţi la intrările şi ieşirile lui P_n şi specifică permutările fiecărei $P_{n/2}$ pentru a realiza permutarea cerută a celor n elemente. Demonstraţi că algoritmul descris este corect.

d. Cât este adâncimea şi dimensiunea reţelei P_n? Cât durează pe o maşină RAM obişnuită calculul tuturor poziţionărilor comutatorilor, incluzând şi pe cele din $P_{n/2}$?

e. Pentru $n > 2$ arătaţi că orice reţea de permutări – nu numai P_n – pentru realizarea permutărilor, necesită două poziţionări distincte ale comutatorilor.

Note bibliografice

Knuth [123] discută, pe larg, reţelele de sortare şi descrie istoricul lor. Se pare că reţelele de sortare au fost discutate prima dată de P. N. Armstrong, R. J. Nelson şi D. J. O'Connor în 1954. La începutul anilor 1960, K. E. Batcher a descoperit prima reţea capabilă să interclaseze două secvenţe de n numere în $O(\lg n)$ unităţi de timp. El a utilizat interclasarea de tip par-impar (problema 28-2), şi, de asemenea, a demonstrat cum poate fi utilizată această tehnică pentru a sorta n numere în $O(\lg^2 n)$ unităţi de timp. Nu peste mult timp, tot el a descoperit un algoritm de sortare bitonă de adâncime $O(\lg n)$, similar celui din secţiunea 28.3. Knuth atribuie principiul zero-unu lui W. G. Bouricius (1954), cel care a demonstrat acest principiu pentru arbori de decizie.

Mult timp a fost o problemă deschisă, dacă există o reţea de sortare de adâncime $O(\lg n)$. În 1983, la această întrebare, s-a dat un răspuns pozitiv, dar nu tocmai satisfăcător. Reţeaua de sortare AKS (numită după autorii ei Ajtai, Komlós şi Szemerédi [8]) poate sorta n numere, având adâncimea $O(\lg n)$ şi folosind $O(n \lg n)$ comparatori. Dar constantele ascunse în O-notaţia sunt foarte mari (multe, multe mii), şi deci metoda, practic, nu poate fi folosită.

29 Circuite aritmetice

Orice model actual de calculator presupune că operaţiile aritmetice de bază: adunare, scădere, înmulţire şi împărţire se pot executa în timp constant. Această ipoteză este rezonabilă deoarece operaţiile efectuate pe o maşină obişnuită solicită consumuri de resurse (costuri) similare. Când se proiectează circuitele pentru aceste operaţii se constată că performanţele lor depind de dimensiunea (numărul de cifre binare ale) operanzilor. De exemplu, se ştie că două numere naturale de câte n cifre zecimale se adună în $\Theta(n)$ paşi (deşi profesorii de obicei nu pun în evidenţă acest număr de paşi).

În acest capitol sunt prezentate circuite care execută operaţii aritmetice. Cu procese seriale, cel mai bun timp asimptotic pentru astfel de operaţii pe n biţi este de $\Theta(n)$. Cu circuite care operează în paralel, se obţin desigur rezultate mai bune. În cele ce urmează, vom prezenta circuite rapide pentru adunare şi înmulţire. (Scăderea este, în esenţă, la fel cu adunarea, iar împărţirea va fi tratată în problema 29-1. Vom presupune că toate datele de intrare sunt numere binare de câte n biţi.

În secţiunea 29.1 prezentăm circuite combinaţionale. Vom constata că adâncimea acestor circuite corespunde cu "timpul lor de execuţie". Sumatorul complet, care este un bloc de bază în cele mai multe circuite prezentate în acest capitol, constituie primul exemplu de circuit combinaţional. În secţiunea 29.2 sunt prezentate două circuite combinaţionale pentru adunare: sumator cu transport propagat, care lucrează în timp $\Theta(n)$, şi sumator cu transport anticipat, care lucrează într-un timp de numai $O(\lg n)$. Se mai prezintă sumatorul cu transport salvat, care reduce problema însumării a trei numere la problema însumării a două numere în timp $\Theta(1)$. În secţiunea 29.3 sunt prezentate două circuite combinaţionale de înmulţire: multiplicatorul matriceal care lucrează într-un timp $\Theta(n)$ şi multiplicatorul arborescent Wallace, care necesită un timp $\Theta(\lg n)$. În sfârşit, secţiunea 29.4 prezintă circuitele cu un registru de memorare a elementelor de ceas şi arată cum se poate reduce hardware-ul necesar reutilizând circuitele combinaţionale.

29.1. Circuite combinaţionale

Similar cu ceea ce s-a prezentat în capitolul 28, circuitele combinaţionale operează în ***paralel***: multe elemente pot fi calculate simultan, într-un singur pas. În această secţiune vom defini circuitele combinaţionale şi vom investiga modul în care circuitele mari pot fi realizate din circuite poartă elementare.

Elemente combinaţionale

În calculatoarele reale, circuitele aritmetice sunt fabricate din elemente combinaţionale conectate între ele prin fire. Un ***element combinaţional*** este orice element de circuit care are un număr constant de intrări şi ieşiri şi efectuează o funcţie bine definită. Unele dintre elementele definite sunt ***elemente combinaţionale booleene***, intrările şi ieşirile lor fiind din mulţimea $\{0, 1\}$, 0 reprezentând FALS şi 1 reprezentând ADEVĂRAT.

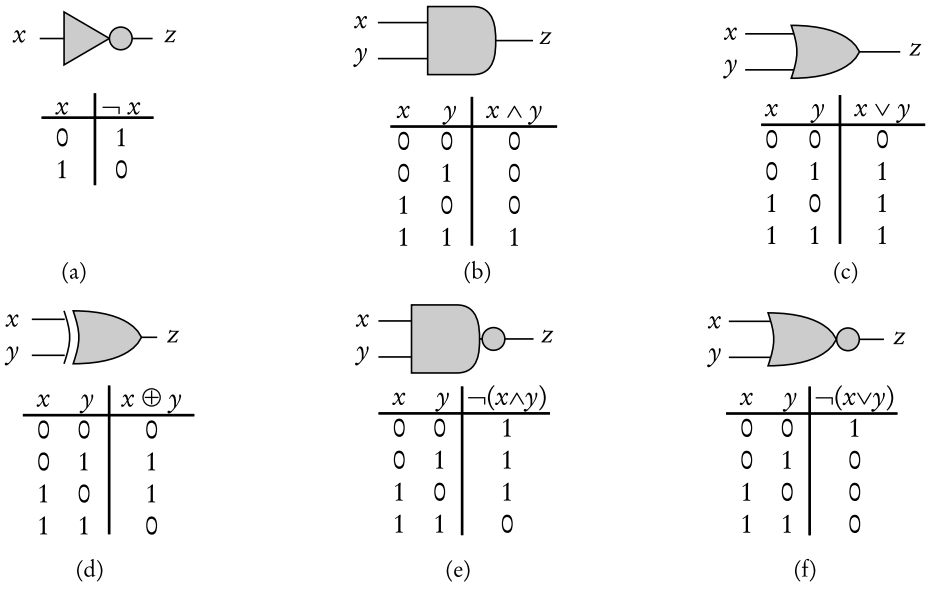


Figura 29.1 Şase porţi logice de bază, cu intrările şi ieşirile binare. Sub fiecare poartă este prezentat tabelul de adevăr care îi descrie funcţionarea. **(a)** Poarta NOT. **(b)** Poarta AND. **(c)** Poarta OR. **(d)** Poarta XOR (SAU-exclusiv). **(e)** Poarta NAND (NOT-AND). **(f)** Poarta NOR (NOT-OR).

Un element combinaţional care calculează valoarea unei funcţii booleene simple este numit ***poartă logică***. În figura 29.1 sunt prezentate cele patru porţi logice care vor fi folosite în acest capitol: ***poarta NOT*** (sau ***inversor***), ***poarta AND***, ***poarta OR*** şi ***poarta XOR***. (Mai sunt prezentate ***porţile NAND*** şi ***NOR***, folosite în unele exerciţii.) Poarta NOT primeşte o singură ***intrare*** binară x, cu una dintre valorile 0 sau 1, şi produce ***ieşirea*** binară z care este opusă intrării. Fiecare dintre celelalte porţi primeşte câte două intrări x şi y şi produce, la ieşire, o valoare binară z.

Modul de operare al fiecărei porţi şi al fiecărui element combinaţional poate fi descris printr-un ***tabel de adevăr***, ilustrat în figura 29.1 sub fiecare poartă. Un astfel de tabel de adevăr produce ieşirea pentru fiecare combinaţie de intrări posibile. De exemplu, tabelul de adevăr pentru funcţia XOR arată că intrarea $x = 0$ şi $y = 1$ produce ieşirea $z = 1$; calculează deci "SAU exclusiv" între cele două valori. Prin simbolul $\neg$ vom nota funcţia NOT, prin $\wedge$ funcţia AND, prin $\vee$ funcţia OR, iar prin $\oplus$ funcţia XOR. Astfel, spre exemplu, $0 \oplus 1 = 1$.

Elementele combinaţionale din circuitele reale nu operează instantaneu. O dată fixate valorile de intrare ale elementelor, acestea devin ***instalate*** (devin ***stabile***): aceasta înseamnă că ele stau în această stare un timp suficient de îndelungat, deci se garantează că valoarea de ieşire este stabilă un anumit timp. Vom numi acest timp ***coadă de propagare***. În acest capitol vom presupune că toate elementele combinaţionale au aşteptarea de propagare constantă.

Circuite combinaţionale

Un ***circuit combinaţional*** constă dintr-una sau mai multe elemente combinaţionale interconectate într-o manieră aciclică. Înterconectările sunt numite ***fire***. Un fir poate să conecteze

ieşirea unui element la intrarea unui alt element combinaţional. Astfel, fiecare fir este o ieşire a unui singur element, dar poate fi intrare pentru unul sau mai multe elemente. Numărul de ieşiri ale unui fir poartă numele de ***evantai de ieşire***. Dacă la un fir nu se conectează nici un element de ieşire, atunci circuitul se cheamă ***circuit de intrare***, de la care sunt acceptate intrări din surse exterioare. În mod similar, dacă la un fir nu este conectat nici un element de intrare, atunci firul este numit ***circuit de ieşire***, care furnizează rezultate pentru destinaţii exterioare. Este, de asemenea, posibil ca un fir interior să furnizeze, prin evantai, intrare spre exterior. Circuitele combinaţionale nu conţin cicluri şi nici nu au memorie (cum ar fi regiştrii descrişi în secţiunea 29.4).

Sumatoare complete

În figura 29.2 este prezentat un circuit combinaţional care realizează ***adunare completă*** pentru o poziţie. Intrarea este compusă din trei biţi: x, y şi z, iar ieşirea este compusă din doi biţi: s şi c, în concordanţă cu următorul tabel de adevăr:

x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Ieşirea s constituie ***bitul de paritate*** pentru intrări (cifra curentă a sumei):

$$s = \text{paritate}(x, y, z) = x \oplus y \oplus z, \tag{29.1}$$

iar ieşirea c este ieşirea de ***majoritate*** (cifra de transport la rangul următor):

$$c = \text{majoritate}(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z). \tag{29.2}$$

(În general, funcţiile de paritate şi majoritate pot avea la intrare un număr oarecare de biţi. Paritatea este 1 dacă şi numai dacă numărul de biţi 1 este impar, iar majoritatea este 1 dacă şi numai dacă mai mult de jumătate din biţi au valoarea 1.) Biţii c şi s, luaţi împreună, realizează suma biţilor x, y, z. De exemplu, dacă $x = 1, y = 0, z = 1$, atunci $\langle c, s\rangle = \langle 10\rangle$,[1] care este reprezentarea în binar a numărului 2, adică suma biţilor x, y, z.

Fiecare dintre intrările x, y, z are un evantai de ieşire egal cu 3. Când operaţia executată de un element combinaţional este una comutativă şi asociativă (aşa cum sunt AND, OR şi XOR), vom numi intrările lui ***evantai de intrare***. Astfel, evantaiul de intrare al fiecărei porţi din figura 29.2 este 2, prin urmare putem înlocui porţile XOR A şi E printr-o singură poartă XOR cu evantai de intrare 3. De asemenea, porţile F şi G pot fi înlocuite printr-o singură poartă OR cu trei intrări.

Pentru a constata modul de funcţionare al sumatorului complet, vom presupune că fiecare poartă operează într-o unitate de timp. În figura 29.2(a) se arată setul de intrări care sunt stabile la momentul 0. Doar porţile A–D au, la intrare, aceste mărimi; după aceea, produc în paralel, la

[1] Pentru claritate, omitem virgulele dintre elementele şirului, atunci când acestea sunt biţi.

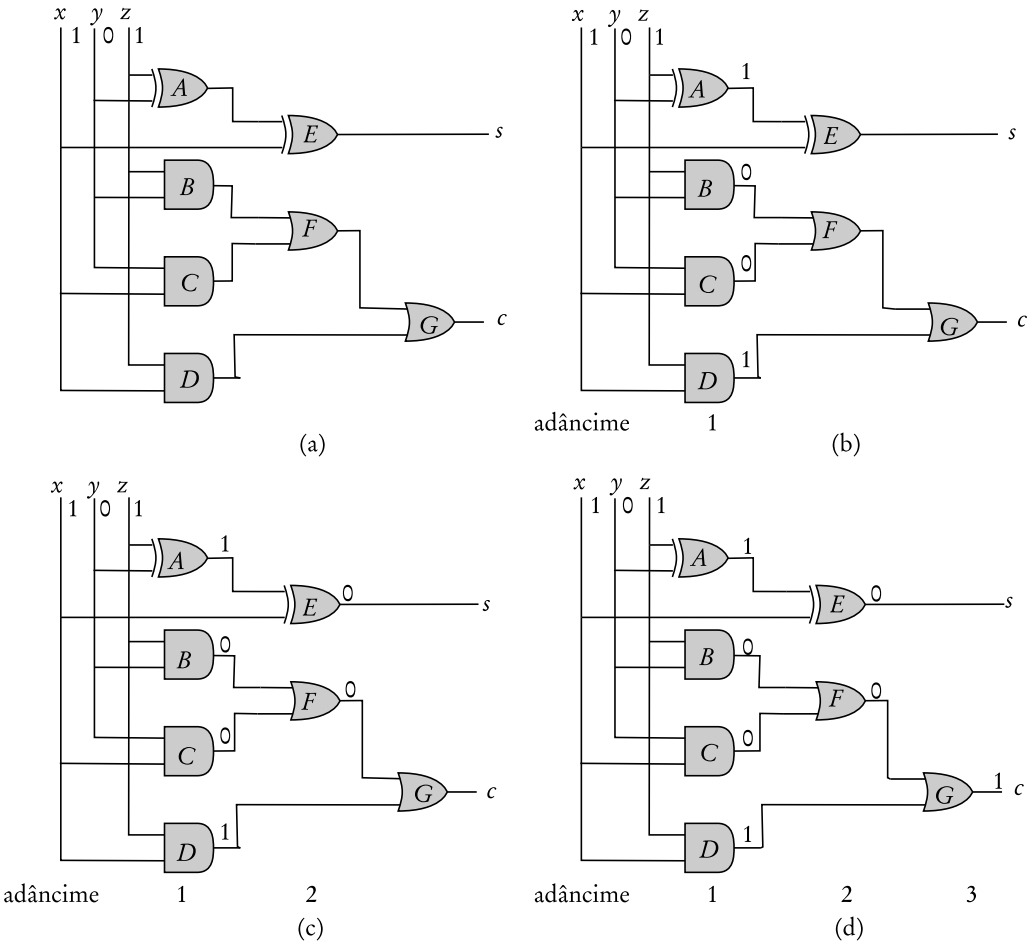


Figura 29.2 Un circuit sumator complet. **(a)** La momentul 0, intrările apar la cele trei fire de intrare. **(b)** La momentul 1, apar valorile la ieşirile din porţile A–D care au adâncimea 1. **(c)** La momentul 2 apar valorile la ieşirile porţilor E şi F de adâncime 2. **(d)** La momentul 3 produce ieşire poarta G şi se realizează ieşirile circuitului.

momentul 1, valorile din figura 29.2(b). Porţile E şi F, dar nu şi G, au la momentul 1, intrarea stabilă şi produc, la momentul 2, valorile din figura 29.2(c). Ieşirea din poarta E este bitul s, deci ieşirea s este gata la momentul 2. Pentru ieşirea c, mai este nevoie de acţiunea porţii G care creează, la momentul 3, valoarea lui c, aşa cum se vede în figura 29.2(d).

Adâncimea circuitului

În acelaşi mod în care s-au comparat reţelele în capitolul 28, măsurăm coada de propagare a circuitului combinaţional în funcţie de cel mai mare număr de circuite combinaţionale aflate pe un drum de la intrări la ieşiri. Astfel, ***adâncimea*** unui circuit este numărul de circuite aflate pe cea mai lungă ramură care compune circuitul. Ea corespunde "timpului de execuţie" în

cazul cel mai defavorabil. Adâncimea firului de intrare este 0. Dacă un circuit combinaţional are elementele de intrare $x_1, x_2, \ldots, x_n$ de adâncimi $d_1, d_2, \ldots, d_n$, atunci adâncimea ieşirii lui este $\max\{d_1, d_2, \ldots, d_n\}+1$. (Adâncimea unui circuit combinaţional este dată de maximul adâncimilor ieşirilor lui). Deoarece sunt interzise ciclurile în circuitele combinaţionale, acest mod de definire este corect.

Dacă fiecare element combinaţional are un timp constant de calcul al ieşirii, atunci coada de propagare este proporţională cu adâncimea lui. În figura 29.2 sunt prezentate adâncimile fiecărei porţi a sumatorului complet. Deoarece poarta cu cea mai mare adâncime este G, circuitul complet are adâncimea 3, proporţională cu timpul cel mai defavorabil de calcul al acestei funcţii.

Uneori, un circuit combinaţional poate să calculeze într-un timp mai bun decât adâncimea lui. Presupunem că un subcircuit AND aşteaptă prima linie în adâncime, dar a doua intrare în AND are valoarea 0. Ieşirea din poarta AND dă, automat, ieşirea 0, independent de valoarea intrării aşteptate. Evident că nu se poate conta întotdeauna pe o astfel de situaţie, motiv pentru care este rezonabil să utilizăm adâncimea ca măsură a timpului de execuţie.

Dimensiune circuit

Noţiunea de ***dimensiune*** este o altă măsură folosită, pe lângă adâncimea circuitului pe care, de regulă, dorim s-o minimizăm în proiectarea circuitelor. Ea reprezintă numărul de elemente combinaţionale conţinute în circuit. De exemplu, sumatorul din figura 29.2 are dimensiunea 7 deoarece foloseşte 7 porţi.

Această măsură se foloseşte rar la circuite mici. Deoarece un sumator are un număr constant de intrări şi de ieşiri şi calculează o funcţie bine definită, acesta corespunde definiţiei circuitului combinaţional. Astfel de circuite pot fi considerate ca având toate dimensiunea 1. De fapt, conform acestei definiţii, *orice* element combinaţional are dimensiunea 1.

Noţiunea este aplicată familiilor de circuite care calculează mai multe funcţii similare. Ne referim, de exemplu, la situaţii când avem circuite sumatoare care adună numere de câte n biţi. Aici nu vorbim despre un singur circuit, ci despre familii de circuite, câte unul pentru fiecare poziţie binară calculată.

În acest context, definirea dimensiunii capătă un sens practic. Este, astfel, permisă definirea elementelor de circuit convenabile fără a afecta dimensiunea cu mai mult de un factor constant. În practică, noţiunea de dimensiune este mai complicată, în ea fiind înglobate atât numărul de elemente, cât şi cerinţele tehnice ce se impun la integrarea elementelor într-un circuit fizic.

Exerciţii

29.1-1 Schimbaţi, în figura 29.2, intrarea y la 1. Prezentaţi valorile ce apar pe fiecare fir.

29.1-2 Arătaţi cum se poate construi un circuit de paritate cu n intrări folosind $n-1$ porţi XOR şi adâncimea $\lceil \lg n \rceil$.

29.1-3 Arătaţi că orice element combinaţional poate fi construit cu un număr constant de porţi AND, OR şi NOT. (*Indicaţie*: Implementaţi tabelul de adevăr al elementului.)

29.1-4 Arătaţi că orice funcţie booleană poate fi construită numai cu porţi NAND.

29.1-5 Construiţi un circuit combinaţional care efectuează funcţia sau-exclusiv, folosind numai patru porţi NAND cu câte două intrări.

8	7	6	5	4	3	2	1	0		i
1	1	0	1	1	1	0	0	0	=	c
	0	1	0	1	1	1	1	0	=	a
	1	1	0	1	0	1	0	1	=	b
1	0	0	1	1	0	0	1	1	=	s

Figura 29.3 Suma a două numere de câte 8 biţi, $a = \langle 01011110 \rangle$ şi $b = \langle 11010101 \rangle$ dă suma lor pe 9 biţi $s = \langle 100110011 \rangle$. Fiecare bit c_i este bit de transport. Fiecare coloană de biţi reprezintă, de sus în jos pentru fiecare i, biţii c_i, a_i, b_i, s_i. Transportul iniţial c_0 este întotdeauna 0.

29.1-6 Fie C un circuit combinaţional cu n intrări, n ieşiri şi adâncimea d. Dacă se conectează două copii ale lui C, ieşirile unuia conectându-se direct la intrările celuilalt, care este adâncimea maximă a tandemului celor două circuite? Dar adâncimea minimă?

29.2. Circuite de sumare

Acum vom trata problema adunării numerelor reprezentate binar. Vom prezenta, în acest, scop trei circuite combinaţionale. Primul este sumatorul cu transport propagat, care poate să adune două numere de câte n biţi în timp $\Theta(n)$ şi utilizând un circuit având o dimensiune $\Theta(n)$. Al doilea este sumatorul cu transport anticipat, de dimensiune $\Theta(n)$ care adună în timp $O(\lg n)$. Al treilea este un circuit cu transport salvat, care, în timp $O(1)$, poate reduce suma a trei numere de câte n biţi la suma unui număr de n biţi şi a unuia de $n + 1$ biţi. Circuitul are dimensiunea $\Theta(n)$.

29.2.1. Sumator cu transport propagat

Vom începe cu cea mai cunoscută metodă de adunare a numerelor binare. Presupunem că numărul nenegativ a este reprezentat în binar printr-o secvenţă de n biţi $\langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, unde $n \geq \lceil \lg(a + 1) \rceil$ şi

$$a = \sum_{i=0}^{n-1} a_i 2^i.$$

Fiind date două numere de câte n biţi $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle, b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, dorim să obţinem o sumă pe $n + 1$ biţi $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$. În figura 29.3 este prezentat un astfel de exemplu de adunare a două numere de câte 8 biţi. Adunarea se efectuează de la dreapta spre stânga, fiecare transport fiind propagat din coloana i în coloana $i + 1$, pentru $i = 0, 1, \ldots, n - 1$. În poziţia i, se adună a_i cu b_i şi cu ***transportul de intrare*** c_i. Rezultatele sunt ***bitul sumă*** s_i şi ***transportul de ieşire*** c_{i+1}, acesta din urmă devenind transport de intrare pentru poziţia $i + 1$. Transportul iniţial este $c_0 = 0$, iar bitul sumă s_n este transportul de ieşire c_n.

Se observă (vezi ecuaţia (29.1)) că bitul sumă s_i reprezintă paritatea biţilor a_i, b_i şi c_i. În plus, transportul de ieşire c_{i+1} reprezintă majoritatea din a_i, b_i şi c_i (vezi ecuaţia (29.2)). Astfel, fiecare etapă a adunării poate fi efectuată cu un sumator complet.

Un sumator cu ***transport propagat***, pe n biţi este format prin legarea în cascadă a n sumatori completi $FA_0, FA_1, \ldots, FA_{n-1}$, (FA - Full Addition) legând transportul de ieşire c_{i+1}

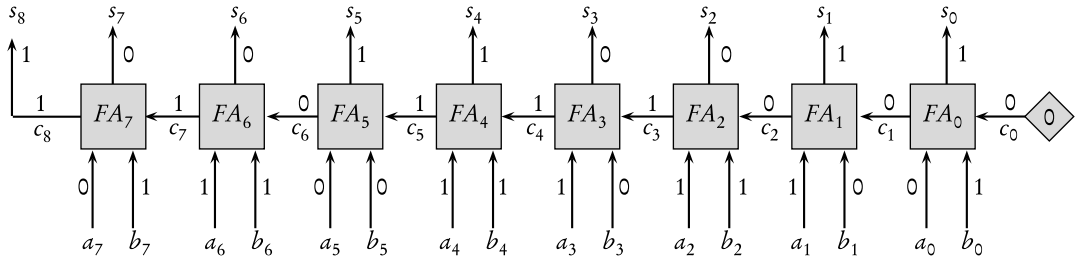


Figura 29.4 Un sumator pe 8 biţi cu transport propagat care efectuează suma din figura 29.3. Bitul de transport c_0 este iniţializat cu 0, iar propagarea transporturilor se face de la dreapta la stânga.

din FA_i cu intrarea de transport din FA_{i+1}. În figura 29.4 este prezentat un sumator de 8 biţi cu transport propagat. Bitul de transport este "propagat" de la dreapta spre stânga. Bitul c_0 este ***fixat hard*** la 0, semnificaţia lui fiind aceea că, iniţial, nu există transport. Cei $n+1$ biţi ai numărului de ieşire sunt $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$, unde s_n este egal cu c_n, transportul de ieşire din FA_n.

Deoarece biţii de transport se propagă de la stânga spre dreapta prin cei n sumatori compleţi, timpul cerut de un sumator cu transport propagat este $\Theta(n)$. Mai exact, sumatorul complet FA_i este la adâncimea $i+1$ în circuit. Deoarece FA_{n-1} este cel mai adânc sumator complet, adâncimea circuitului cu transport propagat este n. Dimensiunea circuitului este $\Theta(n)$ deoarece conţine n elemente combinaţionale.

29.2.2. Sumator cu transport anticipat

Sumatorul cu transport propagat cere un timp $\Theta(n)$ deoarece biţii de transport se propagă prin întregul circuit. Transportul anticipat scurtează acest timp prin accelerarea calculului transporturilor după o schemă arborescentă. Un astfel de sumator poate efectua suma în timp $O(\lg n)$.

Observaţia cheie este aceea că, la sumatorul cu transport propagat, pentru $i \geq 1$, sumatorul complet FA_i are două intrări respectiv pe a_i şi b_i pregătite cu mult timp înainte ca transportul c_i să fie gata. Această observaţie permite exploatarea anticipată a informaţiei parţiale deja prezente.

De exemplu, fie $a_{i-1} = b_{i-1}$. Deoarece transportul c_i este funcţia majoritate, vom avea $c_i = a_{i-1} = b_{i-1}$ *indiferent de valoarea lui* c_{i-1}. Dacă $a_{i-1} = b_{i-1} = 0$, atunci transportul este ***distrus***, deci $c_i = 0$ fără să se mai aştepte sosirea lui c_{i-1}. La fel, dacă $a_{i-1} = b_{i-1} = 1$, atunci transportul este ***generat***, deci $c_i = 1$ fără să se mai aştepte sosirea lui c_{i-1}.

Dacă $a_{i-1} \neq b_{i-1}$, atunci transportul c_i va depinde de c_{i-1}. Mai exact, $c_i = c_{i-1}$ deoarece transportul c_{i-1} decide "votul" pentru bitul majoritar care îl determină pe c_i. În acest caz, transportul este ***propagat*** deoarece transportul de ieşire coincide cu cel de intrare.

În figura 29.5 se rezumă aceste relaţii în termenii ***stării transportului***, unde `k` înseamnă "distrugerea" transportului, `g` înseamnă "generarea" lui, iar `p` înseamnă "propagarea" lui.

Să considerăm un ansamblu format din succesiunea a doi sumatori compleţi: FA_{i-1} urmat de FA_i. Transportul de intrare în ansamblu este c_{i-1}, iar transportul de ieşire din ansamblu este c_{i+1}. Să vedem în ce constau operaţiile de distrugere, generare şi de propagare a transportului pentru acest ansamblu. Operaţia de distrugere are loc fie dacă FA_i distruge transportul, fie dacă FA_{i-1} îl distruge, şi FA_i îl propagă. În mod analog, operaţia de generare are loc dacă FA_i generează transportul sau dacă FA_{i-1} ıl generează, şi FA_i îl propagă. Operaţia de propagare

a_{i-1}	b_{i-1}	c_i	starea de transport
0	0	0	**k**
0	1	c_{i-1}	**p**
1	0	c_{i-1}	**p**
1	1	1	**g**

Figura 29.5 Starea bitului transport de ieşire c_i şi a stării de transport în funcţie de intrările $a_{i-1}, b_{i-1}, c_{i-1}$ din sumatorul complet FA_{i-1}.

		FA_i		
	$\otimes$	**k**	**p**	**g**
FA_{i-1}	**k**	**k**	**k**	**g**
	p	**k**	**p**	**g**
	g	**k**	**g**	**g**

Figura 29.6 Starea transportului la combinarea sumatorilor compleţi FA_{i-1} şi FA_i date de starea transportului operatorului $\oplus$ peste mulţimea {k,p,g}.

$c_{i+1} = c_{i-1}$ are loc dacă ambii sumatori propagă transportul. Tabelul din figura 29.6 rezumă starea transportului pentru acest ansamblu. Este vorba de tabela ***operatorului de stare a transportului*** $\oplus$ peste domeniul {k,p,g}. Asociativitatea este o proprietate importantă a operatorului; verificarea acestei proprietăţi se cere în exerciţiul 29.2-2.

Operatorul de stare poate fi folosit pentru a exprima fiecare bit de transport c_i în funcţie de intrările lui. Să definim $x_0 = \mathtt{k}$ şi

$$x_i = \begin{cases} \mathtt{k} & \text{dacă } a_{i-1} = b_{i-1} = 0, \\ \mathtt{p} & \text{dacă } a_{i-1} \neq b_{i-1}, \\ \mathtt{g} & \text{dacă } a_{i-1} = b_{i-1} = 0, \end{cases} \tag{29.3}$$

pentru $i = 1, 2, \ldots, n$. Astfel, pentru $i = 1, 2, \ldots, n$, valoarea lui x_i este starea transportului din figura 29.5.

Transportul de ieşire c_i din sumatorul complet FA_{i-1} poate să depindă de starea transporturilor din sumatoarele FA_j pentru $j = 0, 1, \ldots, i-1$. Să definim $y_0 = x_0 = \mathtt{k}$ şi

$$y_i = y_{i-1} \otimes x_i = x_0 \otimes x_1 \otimes \ldots \otimes x_i \tag{29.4}$$

pentru $i = 1, 2, \ldots, n$. Ne putem imagina că y_i este un "prefix" al expresiei $x_0 \otimes x_1 \otimes \ldots \otimes x_n$; vom numi procesul de calcul al valorilor $y_0, y_1, \ldots y_i$ ***calcul de prefix***. (Capitolul 30 prezintă aceste calcule de prefixe, într-un context paralel mai general.) În figura 29.7 se arată că valorile x_i şi y_i corespund adunării binare din figura 29.3. Lema următoare exprimă semnificaţia valorilor y_i pentu sumatorul cu transport anticipat.

Lema 29.1 Fie $x_0, x_1, \ldots, x_n$ şi $y_0, y_1, \ldots, y_n$ definite conform ecuaţiilor (29.3) şi (29.4). Pentru $i = 0, 1, \ldots, n$, sunt valabile următoarele afirmaţii:

1. $y_i = \mathtt{k}$ implică $c_i = 0$,

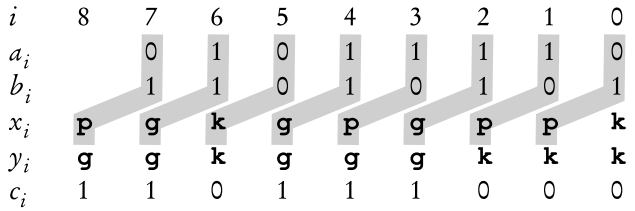


Figura 29.7 Valorile x_i şi y_i, pentru $i = 0, 1, \ldots, 8$ care corespund valorilor a_i, b_i, c_i din adunarea binară din figura 29.3. Fiecare valoare x_i este haşurată împreună cu valorile lui a_{i-1} şi b_{i-1} de care depind.

2. $y_i = \mathtt{g}$ implică $c_i = 1$,
3. $y_i = \mathtt{p}$ nu apare.

Demonstraţie. Demonstraţia este făcută prin inducţie după i. Iniţial, $i = 0$. Avem, prin definiţie, $y_0 = x_0 = \mathtt{k}$ şi $c_0 = 0$. Presupunem că lema este valabilă pentru $i - 1$. În funcţie de valorile lui y_i, se disting trei cazuri.

1. Dacă $y_i = \mathtt{k}$, deoarece $y_i = y_{i-1} \otimes x_i$, definirea operatorului $\otimes$ din figura 29.6 implică ori că $x_i = \mathtt{k}$, ori că $x_i = \mathtt{p}$ şi $y_{i-1} = \mathtt{k}$. Dacă $x_i = \mathtt{k}$, atunci ecuaţia (29.3) implică $a_{i-1} = b_{i-1} = 0$ şi, astfel, $c_i = \text{majoritate}(a_{i-1}, b_{i-1}, c_{i-1}) = 0$. Dacă $x_i = \mathtt{p}$ şi $y_{i-1} = \mathtt{k}$ atunci $a_{i-1} \neq b_{i-1}$, şi, din ipoteza de inducţie, $c_{i-1} = 0$. Astfel, $\text{majoritate}(a_{i-1}, b_{i-1}, c_{i-1}) = 0$ de unde rezultă că $c_i = 0$.
2. Dacă $y_i = \mathtt{g}$, atunci ori avem $x_i = \mathtt{g}$, ori avem $x_i = \mathtt{p}$ şi $y_{i-1} = \mathtt{g}$. Dacă $x_i = \mathtt{g}$ atunci $a_{i-1} = b_{i-1} = 1$, ceea ce implică $c_i = 1$. Dacă $x_i = \mathtt{p}$ şi $y_{i-1} = \mathtt{g}$, atunci $a_{i-1} \neq b_{i-1}$ şi, din ipoteza de inducţie, avem că $c_{i-1} = 1$, ceea ce implică $c_i = 1$.
3. Dacă $y_i = \mathtt{p}$, atunci, din figura 29.6 rezultă că $y_{i-1} = \mathtt{p}$, ceea ce contrazice ipoteza de inducţie. ∎

Lema 29.1 arată că se poate determina fiecare bit de transport c_i calculând fiecare stare de transport y_i. O dată ce s-au calculat toţi biţii de transport, se poate calcula întreaga sumă în timp $\Theta(1)$ prin calcularea în paralel a biţilor sumă $s_i =$ paritate(a_i, b_i, c_i) pentru $i = 0, 1, \ldots, n$ (evident, $a_n = b_n = 0$). Astfel, problema adunării rapide a două numere se reduce la calculul prefixelor de stare a transporturilor $y_0, y_1, \ldots, y_n$.

Calculul prefixelor de stare a transporturilor cu un circuit paralel

Folosind un circuit paralel pentru calculul stărilor transporturilor $y_0, y_1, \ldots, y_n$, sumatorul va fi mult mai rapid decât cel cu transport propagat. Vom prezenta un circuit paralel de calcul al prefixelor cu adâncimea $O(\lg n)$. Circuitul are dimensiunea $\Theta(n)$, ceea ce înseamnă că, asimptotic, are aceeaşi încărcătură hardware ca şi sumatorul cu transport propagat.

Înainte de construcţia circuitului, vom introduce o notaţie care ne va ajuta să înţelegem cum operează circuitul. Pentru numerele întregi i şi j din domeniul $0 \leq i \leq j \leq n$, definim

$$[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$$

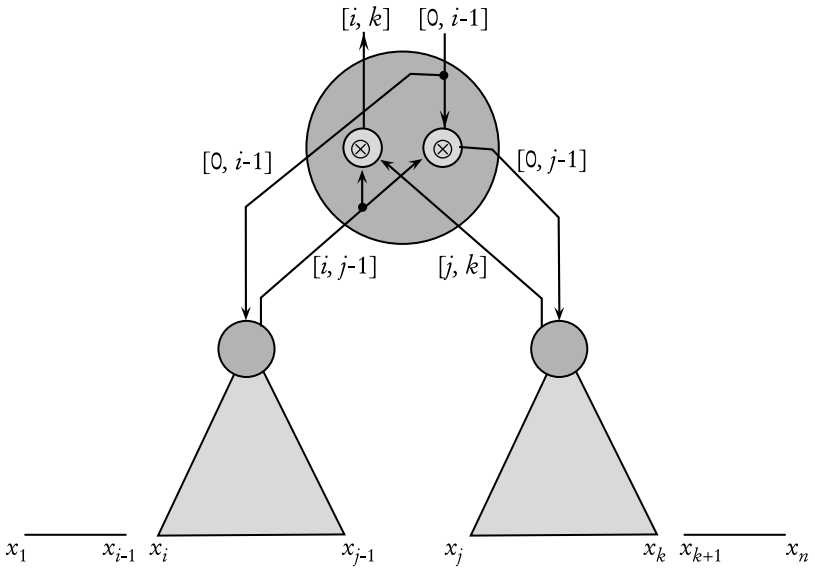


Figura 29.8 Organizarea unui circuit paralel pentru calculul de prefix. Nodul prezentat este rădăcină pentru subarborele care are ca frunze valorile de intrare de la x_i la x_k. Subarborele stâng conţine intrările de la x_i la x_{j-1}, iar subarborele drept pe cele de la x_j la x_k. Nodul constă din două elemente $\otimes$ care operează la momente diferite de timp. Un element calculează $[i,k] \leftarrow [i,j-1] \otimes [j,k]$, iar celălalt element calculează $[0,j-1] \leftarrow [0,i-1] \otimes [i,j-1]$. Pe fire sunt arătate valorile calculate.

Astfel, pentru $i = 0, 1, \ldots, n$, avem $[i,i] = x_i$, deoarece operatorul este idempotent. Pentru i, j, k astfel încât $0 \le i < j \le k \le n$, avem:

$$[i,k] = [i,j-1] \otimes [j,k], \tag{29.5}$$

deoarece operatorul este asociativ. Cu această notaţie, scopul calculului de prefix este să se calculeze $y_i = [0,i]$ pentru $i = 0, 1, \ldots, n$.

Pentru construcţia circuitului, se utilizează numai elemente combinaţionale care evaluează operatorul $\otimes$. În figura 29.8 se arată cum se combină o pereche de elemente $\otimes$ pentru a forma noduri interne într-un arbore binar complet. În figura 29.9 se ilustrează circuitul paralel de prefix pentru $n = 8$. Trebuie remarcat faptul că, deşi firele apar în circuit sub forma unui arbore, circuitul însuşi nu este un arbore, ci un circuit combinaţional pur. Intrările $x_1, x_2, \ldots, x_n$ sunt asociate frunzelor, iar x_0 este în rădăcină. Ieşirile $y_0, y_1, \ldots, y_{n-1}$ sunt obţinute în frunze, iar ieşirea y_n se obţine în rădăcină. (Pentru o înţelegere mai uşoară, în calculul prefixelor din figurile 29.8 şi 29.9 din această secţiune indicii variabilelor cresc de la stânga la dreapta, în loc să crească de la dreapta spre stânga, ca în celelalte figuri.)

De obicei, cele două elemente $\otimes$ din fiecare nod acţionează la momente diferite şi au adâncimi diferite în circuit. Aşa cum se vede în figura 29.8, dacă subarborele cu rădăcina într-un nod dat are intrarea $x_i, x_{i+1}, \ldots, x_k$, subarborele lui stâng are intrarea $x_i, x_{i+1}, \ldots, x_{j-1}$, cel drept $x_j, x_{j+1}, \ldots, x_k$, atunci nodul trebuie să producă, pentru părinţii săi, produsul $[i,k]$ al tuturor intrărilor acoperite de subarborele său. Deoarece putem presupune inductiv că fiii stâng şi drept ai nodului calculează produsele $[i,j-1]$ şi $[j,k]$, nodul utilizează unul dintre cele două elemente

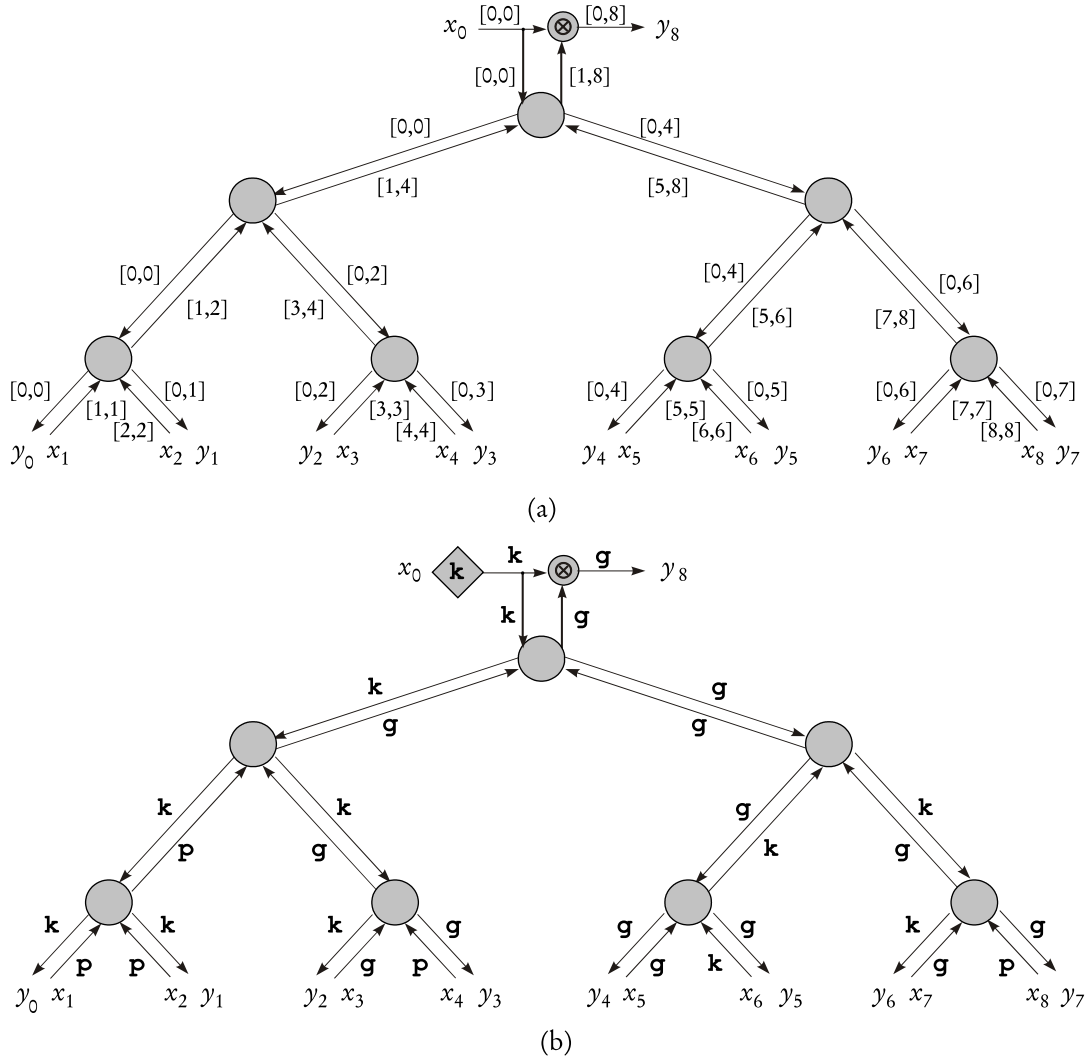


Figura 29.9 Un circuit paralel de prefix pentru $n = 8$. (**a**) Structura completă a circuitului cu valorile de transport pe fiecare fir. (**b**) Acelaşi circuit cu valorile corespunzătoare din figurile 29.3 şi 29.7.

pentru a calcula $[i, k] \leftarrow [i, j-1] \otimes [j, k]$.

La un timp oarecare după prima fază a calculului, nodul primeşte de la părintele lui producţia $[0, i-1]$ cu toate intrările din stânga lui x_i calculate. Apoi nodul calculează valorile pentru fiii lui. Pentru fiul stâng, cel mai din stânga fiu al lui este tot x_i, motiv pentru care îi trimite neschimbate valorile $[0, i-1]$. Pentru fiul din dreapta cel mai din stânga fiu este x_j, deci acestuia îi transmite valorile $[0, j-1]$. Deoarece nodul primeşte valorile $[0, i-1]$ de la părintele lui şi valorile $[i, j-1]$ de la fiul lui stâng, el calculează doar $[0, j-1] \leftarrow [0, i-1] \otimes [i, j-1]$ şi le trimite la fiul drept.

În figura 29.9 sunt prezentate rezultatele circuitului, inclusiv marginile care ajung la rădăcină. Valoarea $x_0 = [0, 0]$ este furnizată ca intrare la rădăcină, iar pentru a calcula valoarea $y_n =$

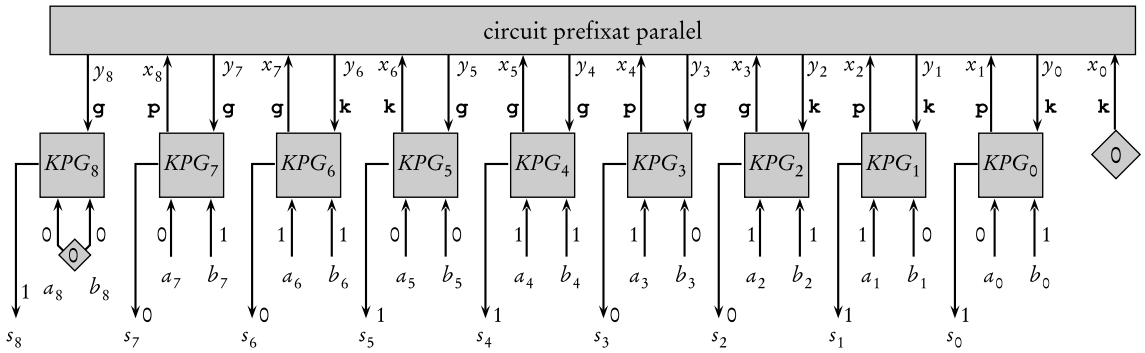


Figura 29.10 Construcţia unui sumator cu transport anticipat de n biţi, prezentat pentru $n = 8$. El constă din $n+1$ cutii KPG_i, $i = 0, 1, \ldots, n$. Fiecare cutie KPG_i are intrările externe a_i, b_i cu a_n, b_n fixaţi la 0 şi calculează starea x_{i+1} a transportului. Valoarea este introdusă într-un circuit prefix paralel care, la rândul lui, returnează prefixul calculat y_i. Apoi, fiecare cutie KPG_i primeşte la intrare valoarea y_i, o interpretează ca transport de intrare c_i, după care produce, la ieşire, bitul sumă $s_i = \text{paritate}(a_i, b_i, c_i)$. Valorile binare indicate pe săgeţi coincid cu cele din figura 29.3 şi 29.9.

$[0, n] = [0, 0] \otimes [1, n]$ se folosesc, în general, unul sau mai multe elemente $\otimes$.

Dacă n este o putere a lui 2, atunci circuitul paralel de prefix foloseşte $2n - 1$ elemente $\otimes$. El consumă doar un timp $O(\lg n)$ pentru a calcula cele $n + 1$ prefixe deoarece calculul merge sus şi jos în arbore. Exerciţiul 29.2-5 studiază, în detaliu, adâncimea circuitului.

Sumator complet cu transport anticipat

După ce am prezentat circuitul paralel prefix, putem completa descrierea sumatorului complet cu transport anticipat. Figura 29.10 prezintă construcţia. Un ***sumator cu transport anticipat*** de n biţi constă din $n + 1$ ***cutii KPG*** (denumire preluată de la mulţimea $\{\texttt{k},\texttt{p},\texttt{g}\}$ peste care operează $\otimes$), fiecare având dimensiunea $\Theta(1)$ şi un circuit paralel de prefix cu intrările $x_0, x_1, \ldots, x_n$ (x_0 este fixat hard la `k`) şi ieşirile $y_0, y_1, \ldots, y_n$. Cutia KPG_i primeşte intrările externe a_i şi b_i şi produce bitul sumă s_i. (Biţii de intrare a_n, b_n sunt 0.) Fiind date a_{i-1}, b_{i-1}, cutia KPG_{i-1} calculează $x_i \in \{\texttt{k},\texttt{p},\texttt{g}\}$ conform ecuaţiei (29.3) şi trimite valorarea obţinută la intrarea x_i a circuitului prefix paralel (valoarea x_{n+1} este ignorată). Toate calculele pentru x_i sunt efectuate consumând un timp $\Theta(1)$. După o aşteptare de $O(\lg n)$, circuitul prefix paralel produce ieşirea $y_0, y_1, \ldots, y_n$. Conform lemei 29.1, y_i este fie `k`, fie `g`, dar niciodată `p`. Fiecare valoare y_i indică transportul de intrare în sumatorul complet FA_i a sumatorului cu transport propagat: $y_i = \texttt{k}$ implică $c_i = 0$ şi $y_i = \texttt{g}$ implică $c_i = 1$. Valoarea y_i indică, pentru KPG_i, transportul de intrare c_i în timp constant, iar bitul sumă $s_i = \text{paritate}(a_i, b_i, c_i)$ este produs, de asemenea, în timp constant. Astfel, sumatorul cu transport anticipat operează în timp $O(\lg n)$ şi are dimensiunea $\Theta(n)$.

29.2.3. Sumator cu transport salvat

Sumatorul cu transport anticipat adună două numere de câte n biţi în timp $O(\lg n)$. S-ar putea să pară surprinzător, dar adunarea a trei numere de câte n biţi necesită, în plus, numai

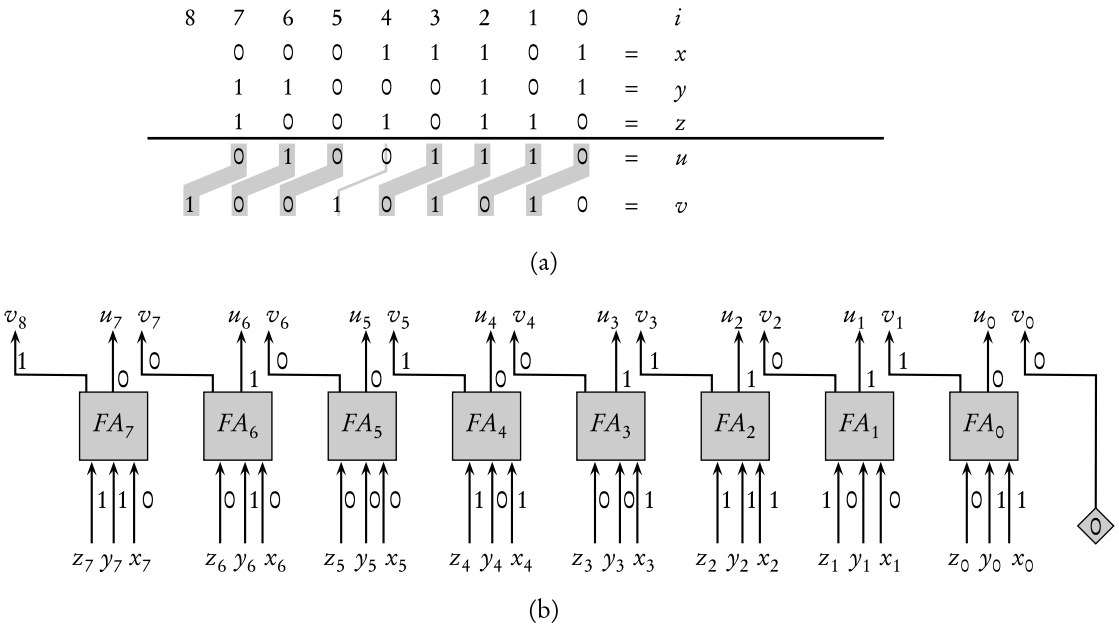


Figura 29.11 **(a)** Adunare cu salvare transport. Fiind date trei numere de câte n biţi x, y şi z, se obţine un număr u de n biţi şi un număr v de $n+1$ biţi, astfel încât $x+y+z=u+v$. A i-a pereche haşurată este obţinută în funcţie de x_i, y_i şi z_i. (**(b)**) Un sumator cu transport salvat, pe 8 biţi. Fiecare sumator complet FA_i primeşte, la intrare, x_i, y_i şi z_i şi obţine, la ieşire, bitul sumă u_i şi transportul de ieşire v_{i+1}. Bitul $v_0=0$.

un timp constant. Astfel se reduce problema adunării a trei numere la adunarea a numai două numere.

Fiind date trei numere de câte n biţi $x=\langle x_{n-1},x_{n-2},\ldots,x_0\rangle$, $y=\langle y_{n-1},y_{n-2},\ldots,y_0\rangle$ şi $z=\langle z_{n-1},z_{n-2},\ldots,z_0\rangle$, un ***sumator cu transport salvat*** obţine un număr de n biţi $u=\langle u_{n-1},u_{n-2},\ldots,u_0\rangle$ si un număr de $n+1$ biţi $v=\langle v_n,v_{n-1},\ldots,v_0\rangle$, astfel încât

$$u+v=x+y+z.$$

Aşa cum se vede în figura 29.11(a), calculul rezultă din:

$$\begin{aligned} u_i &= \text{paritate}(x_i,y_i,z_i),\\ v_{i+1} &= \text{majoritate}(x_i,y_i,z_i),\end{aligned}$$

pentru $i=0,1,\ldots,n-1$. Bitul v_0 este egal întotdeauna cu 0.

Sumatorul din figura 29.11(b) constă din n sumatori completi $FA_0,FA_1,\ldots,FA_{n-1}$. Pentru $i=0,1,\ldots,n-1$, sumatorul complet are intrările x_i, y_i şi z_i. Bitul sumă de ieşire al lui FA_i este u_i, iar transportul de ieşire al lui FA_i este v_{i+1}. Bitul $v_0=0$.

Deoarece calculele celor $2n+1$ biţi de ieşire sunt independente, ele pot fi efectuate în paralel. Astfel, sumatorul cu transport salvat operează în timp $\Theta(1)$ şi are dimensiunea $\Theta(n)$. De aceea, pentru suma a trei numere de câte n biţi efectuată de sumatorul cu transport salvat, este necesar un timp de $\Theta(1)$ pentru a ajunge la suma a două numere. Apoi, un sumator cu transport anticipat, în timp $O(\lg n)$ adună cele două numere. Această metodă nu este mai bună asimptotic

decât cea cu transport anticipat, dar este mai rapidă în practică. Mai mult, aşa cum se va vedea în secţiunea 29.3, acest algoritm este nucleul unui algoritm rapid de multiplicare.

Exerciţii

29.2-1 Fie $a = \langle 01111111 \rangle$, $b = \langle 00000001 \rangle$ şi $n = 8$. Determinaţi biţii de sumă şi transport a ieşirilor din sumatorii completeţi la aplicarea sumatorului cu transport propagat. Găsiţi stările de transport $x_0, x_1, \ldots, x_8$ corespunzătoare lui a şi b, apoi etichetaţi fiecare fir al circuitului de calcul prefix din figura 29.9 cu aceste valori. Determinaţi rezultatele de ieşire $y_0, y_1, \ldots, y_8$.

29.2-2 Demostraţi că operatorul $\otimes$ dat în figura 29.5 este asociativ.

29.2-3 Exemplificaţi cum se poate construi un circuit prefix paralel care lucrează în timp $O(\lg n)$ atunci când n nu este o putere exactă a lui 2, desenând un circuit prefix paralel pentru $n = 11$. Caracterizaţi performanţele unui astfel de circuit (când arborele lui binar este unul oarecare).

29.2-4 Prezentaţi, la nivel de porţi, construcţia unei cutii KPG_i. Presupuneţi că fiecare ieşire x_i este reprezentată prin $\langle 00 \rangle$ dacă $x_i = \mathtt{k}$, prin $\langle 11 \rangle$ dacă $x_i = \mathtt{g}$, prin $\langle 01 \rangle$ sau $\langle 10 \rangle$ dacă $x_i = \mathtt{p}$. De asemenea, presupuneţi că fiecare intrare y_i este reprezentată prin 0 dacă $y_i = \mathtt{k}$, respectiv prin 1 dacă $y_i = \mathtt{g}$.

29.2-5 Etichetaţi fiecare fir din circuitul prefix din figura 29.9(a) cu adâncimea lui. Un ***drum critic*** într-un circuit este drumul cu cel mai mare număr de elemente combinaţionale dintre toate drumurile de la intrare la ieşire. Identificaţi drumul critic din figura 29.9(a) şi arătaţi că lungimea lui este $O(\lg n)$. Arătaţi că unele noduri au elemente $\otimes$ care operează în timp $\Theta(\lg n)$. Există noduri ale căror elemente $\otimes$ operează simultan?

29.2-6 Fie o diagramă bloc recursivă de tipul circuitului din figura 29.12, pentru orice număr n care este o putere a lui 2. Arătaţi că această diagramă bloc este un circuit care efectuează un calcul de prefix. Arătaţi că adâncimea circuitului este $\Theta(\lg n)$, iar dimensiunea lui este $\Theta(n \lg n)$.

29.2-7 Care este evantaiul maxim de ieşire pentru firele dintr-un sumator cu transport anticipat? Arătaţi că suma se poate calcula în timp $O(\lg n)$ cu un circuit de dimensiune $\Theta(n)$ dacă se impune restricţia ca porţile să aibă un evantai de ieşire de $O(1)$.

29.2-8 Un ***circuit înalt*** are n intrări binare şi $m = \lceil \lg(n+1) \rceil$ ieşiri. Interpretată ca număr binar, ieşirea reprezintă numărul de cifre 1 din intrare. De exemplu, dacă intrarea este $\langle 10011110 \rangle$, ieşirea este $\langle 101 \rangle$, ceea ce arată că, în intrare, există 5 cifre 1. Descrieţi un circuit înalt de adâncime $O(\lg n)$ şi de dimensiune $\Theta(n)$.

29.2-9 $\star$ Arătaţi că o adunare pe n biţi poate fi realizată cu un circuit combinaţional de adâncime 4 şi dimensiunea polinomială în n, dacă pentru porţile AND şi OR sunt permise evantaie de intrare de orice mărime. (*Opţional*: Realizaţi adâncimea 3).

29.2-10 $\star$ Presupunem că, folosind un sumator cu transport propagat, se însumează două numere aleatoare de câte n biţi, unde fiecare bit, independent de ceilalţi poate primi, cu aceeaşi probabilitate, valoarea 0 sau 1. Arătaţi că, cu o probabilitate de cel puţin $1 - 1/n$, nu se propagă transport mai departe decât $O(\lg n)$ stări consecutive. Cu alte cuvinte, deşi adâncimea sumatorului cu transport propagat este $\Theta(n)$, pentru două numere oarecare, ieşirea se obţine, în cele mai multe cazuri, în timp de maximum $O(\lg n)$.

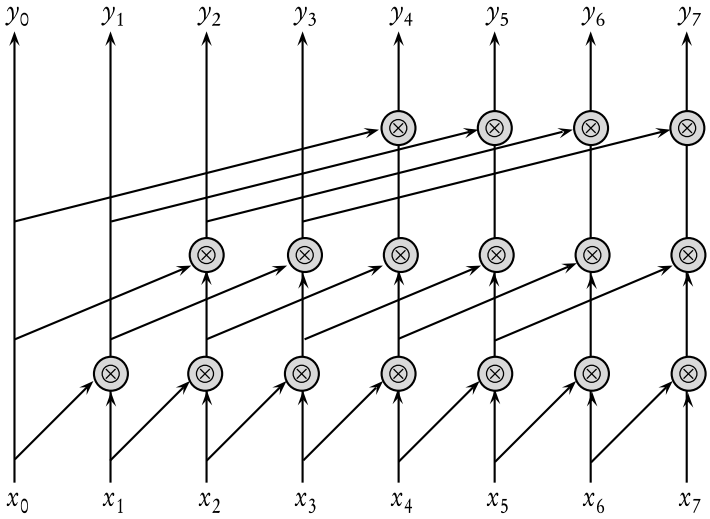


Figura 29.12 Un circuit prefix, folosit în exerciţiul 29.2-6.

29.3. Circuite multiplicatoare

Algoritmul clasic de înmulţire din figura 29.13 poate calcula produsul de $2n$ biţi $p = \langle p_{2n-1}, p_{2n-2}, \ldots, p_0 \rangle$ a două numere de câte n biţi $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ şi $b = \langle b_{n-1}, b_{n-2}, \ldots, p_0 \rangle$. Se examinează fiecare bit al lui b, de la b_0 la b_{n-1}. Pentru fiecare bit b_i având valoarea 1, se adună la produs a deplasat la stânga cu i poziţii. Pentru fiecare bit b_i cu valoarea 0, se adună la produs 0. Astfel, notând $m^{(i)} = a \cdot b_i \cdot 2^i$, se calculează:

$$p = a \cdot b = \sum_{i=0}^{n-1} m^{(i)}.$$

Fiecare termen $m^{(i)}$ este numit ***produs parţial***. Există n produse parţiale având biţi în poziţiile de la 0 la $2n - 2$ care se însumează. Transportul de ieşire de la bitul cel mai semnificativ se constituie în bitul final din poziţia $2n - 1$.

În această secţiune vom examina două circuite pentru înmulţirea a două numere de câte n biţi. Multiplicatorul matriceal operează în timp $\Theta(n)$ şi are dimensiunea de $\Theta(n^2)$. Multiplicatorul arbore-Wallace are dimensiunea tot $\Theta(n^2)$, dar el operează doar în timp $\Theta(\lg n)$. Ambele circuite se bazează pe algoritmul clasic de înmulţire.

29.3.1. Multiplicator matriceal

Un multiplicator matriceal constă, conceptual, din trei părţi. În prima parte sunt calculate produsele parţiale. Partea a doua însumează produsele parţiale, în acest scop folosind sumatoare cu transport salvat. Partea a treia adună cele două numere obţinute de sumatoarele cu transport salvat folosind fie sumator cu transport propagat, fie sumator cu transport anticipat.

				1	1	1	0	=	a
				1	1	0	1	=	b
				1	1	1	0	=	$m^{(0)}$
			0	0	0	0		=	$m^{(1)}$
		1	1	1	0			=	$m^{(2)}$
	1	1	1	0				=	$m^{(3)}$
1	0	1	1	0	1	1	0	=	p

Figura 29.13 Metoda clasică de înmulţire a lui $a = \langle 1110 \rangle$ cu $b = \langle 1101 \rangle$ şi obţinerea rezultatului $p = \langle 10110110 \rangle$. Vom calcula suma $\sum_{i=0}^{n-1} m^{(i)}$, unde $m^{(i)} = a \cdot b_i \cdot 2^i$ şi $n = 4$. Fiecare termen $m^{(i)}$ este format prin deplasarea lui a, (dacă $b_i = 1$) sau a lui 0 (dacă $b_i = 0$), cu i poziţii la stânga. Biţii care nu se văd sunt 0, indiferent de valorile lui a şi b.

În figura 29.14 este prezentat un ***multiplicator matriceal*** pentru două numere $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ şi $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Valorile a_j sunt propagate vertical, iar valorile b_i sunt propagate orizontal. Fiecare bit de intrare se distribuie la n porţi AND pentru a forma produsele parţiale. Sumatoarele complete, organizate ca sumatoare cu transport salvat, însumează produsele parţiale. Biţii de ordin minim din produsul final dau ieşirea spre dreapta, iar cei de ordin maxim sunt obţinuţi la ieşirile orizontale din partea de jos de la ultimul sumator cu transport salvat.

Să examinăm mai atent construcţia multiplicatorului matriceal. Fiind date la intrare două numere $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ şi $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, biţii produselor parţiale se calculează uşor. Astfel, pentru fiecare $i, j = 0, 1, \ldots, n-1$, avem

$$m_{j+i}^{(i)} = a_j \cdot b_i .$$

Deoarece valorile produselor de câte un bit se calculează direct printr-o poartă AND, toţi biţii produselor parţiale (cu excepţia celor care se ştie că vor fi 0 şi nu trebuie calculaţi) pot fi calculaţi într-un singur pas, folosind n^2 porţi AND.

În figura 29.15 se prezintă modul în care sunt executate însumările cu salvări de transporturi la însumarea produselor parţiale ale produsului din figura 29.13. Execuţia începe cu însumarea produselor parţiale $m^{(0)}$, $m^{(1)}$ şi 0, producând numerele de câte $n+1$ biţi $u^{(1)}$ şi $v^{(1)}$. (Numărul $v^{(1)}$ are $n+1$ biţi şi nu $n+2$, deoarece primii $n+1$ biţi din $m^{(0)}$ şi din 0 sunt 0.) Mai departe, $m^{(0)} + m^{(1)} = u^{(1)} + v^{(1)}$. Apoi, sumatorul cu transport salvat adună $u^{(1)}$, $v^{(1)}$ şi $m^{(2)}$, producând două numere de câte $n+2$ biţi şi anume pe $u^{(2)}$ şi $v^{(2)}$. (Şi aici, $v_{(2)}$ are numai $n+2$ biţi deoarece atât $u_{n+2}^{(1)}$ cât şi $v_{n+2}^{(1)}$ sunt 0.) Apoi, avem $m^{(0)} + m^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$. Înmulţirea continuă însumând cu transport salvat valorile $u^{(i-1)}$, $v^{(i-1)}$ şi $m^{(i)}$, pentru $i = 2, 3, \ldots, n-1$. Rezultatul constă din două numere $u^{(n-1)}$ şi $v^{(n-1)}$ a câte $2n-1$ biţi, unde:

$$u^{(n-1)} + v^{(n-1)} = \sum_{i=0}^{n-1} m^{(i)} = p .$$

De fapt, adunările cu transport salvat din figura 29.15 operează puţin mai mult decât strictul necesar. Se observă că, pentru $i = 1, 2, \ldots, n-1$ şi $j = 0, 1, \ldots, i-1$, avem $m_j^{(i)} = 0$ deoarece

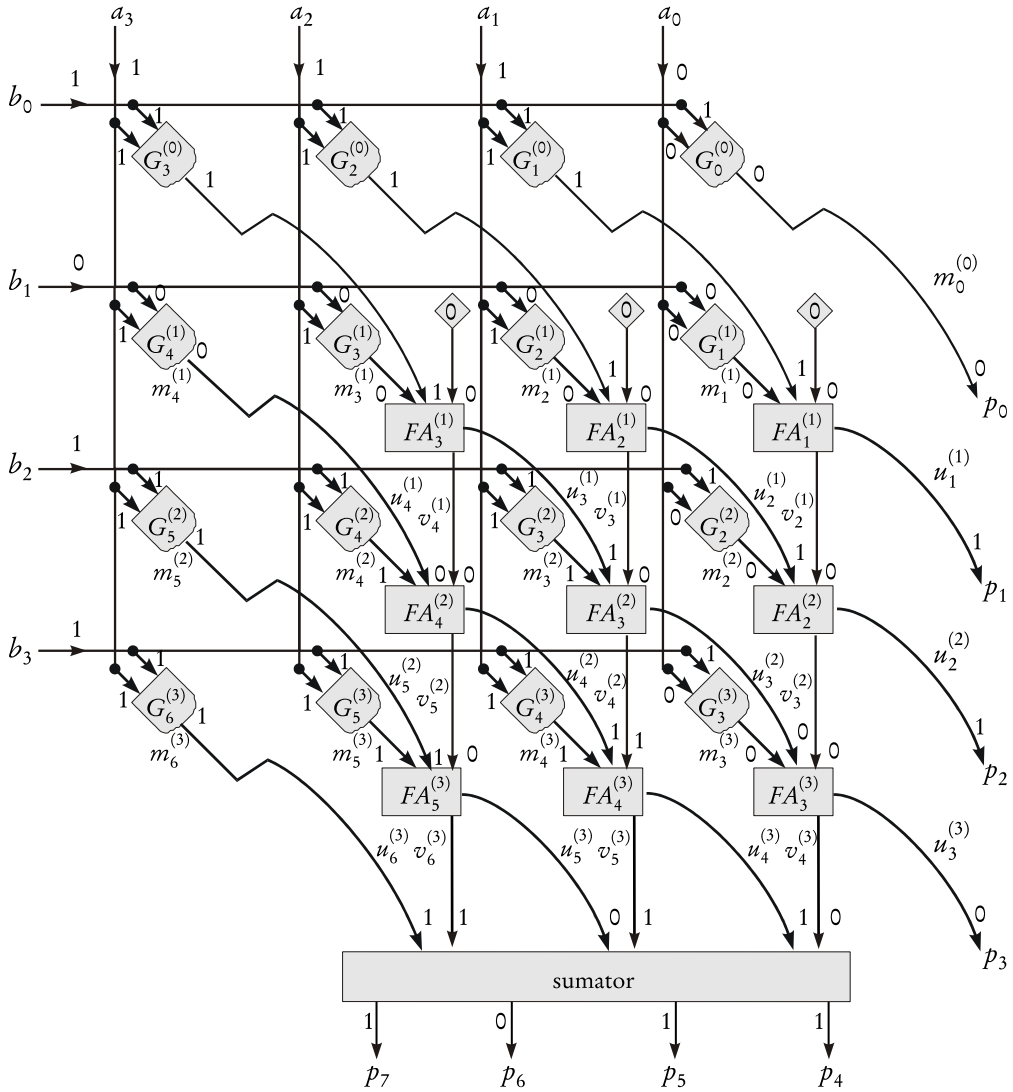


Figura 29.14 Un multiplicator matriceal care calculează produsul $p = \langle p_{2n-1}, p_{2n-2}, \ldots, p_0 \rangle$ a două numere $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ şi $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, arătate aici pentru $n = 4$. Fiecare poartă AND $G_j^{(i)}$ calculează bitul produs parţial $m_j^{(i)}$. Fiecare linie de sumare completă constituie un sumator cu salvare transport. Cei n biţi de rang minim ai produsului sunt $m_0^{(0)}$, iar biţii u se văd în coloana cea mai din dreapta a sumatoarelor complete. Cei n biţi de rang maxim ai produsului sunt formaţi prin adunarea biţilor lui u şi v din linia de jos a sumatoarelor complete. Se observă valorile biţilor de intrare $a = \langle 1110 \rangle$ şi $b = \langle 1101 \rangle$, precum şi ai produsului $p = \langle 10110110 \rangle$, în concordanţă cu figurile 29.13 şi 29.15.

$$
\begin{array}{cccccccccc}
 & & & 0 & 0 & 0 & 0 & = & 0 & \\
 & & & 1 & 1 & 1 & 0 & = & m^{(0)} & \\
 & & 0 & 0 & 0 & 0 & & = & m^{(1)} & \\
\hline
 & & 0 & 1 & 1 & 1 & 0 & = & u^{(1)} & \\
 & & 0 & 0 & 0 & & & = & v^{(1)} & \\
 & 1 & 1 & 1 & 0 & & & = & m^{(2)} & \\
\hline
 & 1 & 1 & 0 & 1 & 1 & 0 & = & u^{(2)} & \\
 & 0 & 1 & 0 & & & & = & v^{(2)} & \\
1 & 1 & 1 & 0 & & & & = & m^{(3)} & \\
\hline
 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & = & u^{(3)} \\
 & 1 & 1 & 0 & & & & & = & v^{(3)} \\
\hline
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & = & p
\end{array}
$$

Figura 29.15 Evaluarea sumelor de produse parţiale prin adunări repetate cu transport salvat. Pentru acest exemplu, $a = \langle 1110 \rangle$ şi $b = \langle 1101 \rangle$. Biţii în locul cărora este spaţiu sunt 0 indiferent de valorile lui a şi b. Se evaluează, mai întâi, $m^{(0)} + m^{(1)} + 0 = u^{(1)} + v^{(1)}$, apoi se evaluează $u^{(1)} + v^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$, apoi $u^{(2)} + v^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$ şi, în final, $p = m^{(0)} + m^{(1)} + m^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$. Se vede că $p_0 = m_0^{(0)}$ şi $p_i = u_i^{(i)}$ pentru $i = 1, 2, \ldots, n-1$.

s-au făcut deplasări ale produselor parţiale. Se observă, de asemenea, că $v_j^{(i)} = 0$, pentru $i = 1, 2, \ldots, n-1$ şi $j = 0, 1, \ldots, i, i+n, i+n+1, \ldots, 2n-1$ (vezi exerciţiul 29.3-1). Din această cauză, fiecare adunare cu transport salvat operează doar pe $n-1$ biţi.

Să urmărim corespondenţa dintre multiplicatorul matriceal şi schema de adunări repetate cu salvare transport. Fiecare poartă AND este etichetată prin $G_j^{(i)}$ pentru indicii i şi j din domeniile $0 \le i \le n-1$ şi $0 \le j \le 2n-2$. Poarta $G_j^{(i)}$ produce $m_j^{(i)}$, adică cel de-al j-lea bit din cel de-al i-lea produs parţial. Pentru $i = 0, 1, \ldots, n-1$, porţile AND din linia i calculează cei mai semnificativi n biţi din produsul parţial $m^{(i)}$, adică $\langle m_{n+i-1}^{(i)}, m_{n+i-2}^{(i)}, \ldots, m_i^{(i)} \rangle$.

Cu excepţia sumatorilor completi din linia de sus (adică pentru $i = 2, 3, \ldots, n-1$), fiecare sumator complet $FA_j^{(i)}$ are trei biţi de intrare: $m_j^{(i)}$, $u_j^{(i-1)}$ şi $v_j^{(i-1)}$ şi produce, la ieşire, doi biţi: $u_j^{(i)}$ şi $v_{j+1}^{(i)}$. (Să reţinem că în coloana cea mai din stânga a sumatorilor completi, $u_{i+n-1}^{(i-1)} = m_{i+n-1}^{(i)}$.) Fiecare $FA_j^{(i)}$ din linia de sus are intrările: $m_j^{(0)}$, $m_j^{(1)}$ şi 0 şi produce biţii de ieşire $u_j^{(1)}, v_{j+1}^{(1)}$.

În final, să examinăm ieşirile multiplicatorului matriceal. Aşa cum s-a observat mai sus, $v_j^{(n-1)} = 0$ pentru $j = 0, 1, \ldots, n-1$. Astfel, $p_j = u_j^{(n-1)}$ pentru $j = 0, 1, \ldots, n-1$. Mai mult, deoarece $m_0^{(1)} = 0$, avem $u_0^{(1)} = m_0^{(0)}$; deoarece cei mai nesemnificativi i biţi din fiecare $m^{(i)}$ şi $v^{(i-1)}$ sunt 0, avem $u_j^{(i)} = u_j^{(i-1)}$ pentru $i = 2, 3, \ldots, n-1$ şi $j = 0, 1, \ldots, i-1$. Astfel, $p_0 = m_0^{(0)}$ şi, prin inducţie, rezultă că $p_i = u_i^{(i)}$ pentru $i = 1, 2, \ldots, n-1$. Biţii produs $\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle$ se obţin dintr-un sumator de n biţi care adună ieşirile din ultima linie a sumatorului complet:

$$\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle = \langle u_{2n-2}^{(n-1)}, u_{2n-3}^{(n-1)}, \ldots, u_n^{(n-1)} \rangle + \langle v_{2n-2}^{(n-1)}, v_{2n-3}^{(n-1)}, \ldots, v_n^{(n-1)} \rangle .$$

Analiză

Într-un multiplicator matriceal datele evoluează din stânga sus spre dreapta jos. El consumă $\Theta(n)$ unităţi de timp pentru a calcula biţii produs de ordin mic $\langle p_{n-1}, p_{n-2}, \ldots, p_0 \rangle$ şi consumă $\Theta(n)$ unităţi de timp pentru adunarea lor completă. Dacă sumatorul este cu transport propagat, el consumă încă $\Theta(n)$ unităţi de timp pentru biţii produs de ordin mare $\langle p_{2n-1}, p_{2n-2}, \ldots, p_n \rangle$. Dacă sumatorul este cu transport anticipat, consumă numai $\Theta(\lg n)$ unităţi de timp, deci timpul total rămâne $\Theta(n)$.

Există n^2 porţi AND şi $n^2 - n$ sumatori completi în multiplicatorul matriceal. Sumatorul pentru biţii de ieşire de ordin înalt contribuie cu încă alte $\Theta(n)$ porţi. Astfel, multiplicatorul matriceal are dimensiunea de $\Theta(n^2)$.

29.3.2. Multiplicatori arbore Wallace

Un ***arbore Wallace*** este un circuit care reduce problema adunării a n numere de n biţi la problema adunării a două numere a câte $\Theta(n)$ biţi. El realizează acest lucru folosind în paralel $\lfloor n/3 \rfloor$ sumatori cu salvare transport spre a converti suma a n numere la suma a $\lceil 2n/3 \rceil$ numere. Apoi construieşte recursiv un arbore Wallace pe cele $\lceil 2n/3 \rceil$ numere rezultate. În acest mod, se reduce progresiv mulţimea de numere până când se ajunge doar la două numere. Efectuând, în paralel, mai multe adunări cu salvare de transport, arborii Wallace permit ca numerele de n biţi să fie înmulţite în $\Theta(\lg n)$ unităţi de timp folosind un circuit de dimensiune $\Theta(n^2)$.

În figura 29.16 este prezentat un arbore Wallace[2] care adună opt produse parţiale $m^{(0)}$, $m^{(1)}, \ldots, m^{(7)}$. Produsul parţial $m^{(i)}$ conţine $n + i$ biţi. Fiecare linie conţine un număr cu toţi biţii lui, nu doar un singur bit. În dreptul fiecărei linii, se specifică pe câţi biţi se reprezintă numărul de pe linia respectivă (vezi exerciţiul 29.3-3). Sumatorul cu transport anticipat din partea de jos adună un număr de $2n - 1$ biţi cu un număr de $2n$ biţi şi calculează produsul de $2n$ biţi.

Analiză

Timpul cerut de un arbore Wallace depinde de adâncimea sumatoarelor cu transport salvat. La fiecare nivel din arbore, fiecare grup de trei numere este redus la două numere, ca în cazul lui $m^{(6)}$ şi $m^{(7)}$ de pe primul nivel. Astfel, adâncimea maximă $D(n)$ a unui sumator cu transport salvat cu n intrări într-un arbore Wallace, este dată de următoarea formulă de recurenţă:

$$D(n) = \begin{cases} 0 & \text{dacă } n \leq 2, \\ 1 & \text{dacă } n = 3, \\ D(\lceil 2n/3 \rceil) + 1 & \text{dacă} n \geq 4, \end{cases}$$

care are soluţia $D(n) = \Theta(\lg n)$, pe baza cazului 2 din teorema master 4.1. Fiecare sumator cu transport salvat cere un timp $\Theta(1)$. Toate cele n produse parţiale pot fi formate în paralel, consumându-se $\Theta(1)$ unităţi de timp. (Primii $i - 1$ biţi nesemnificativi de intrare ai lui $m^{(i)}$, pentru $i = 1, 2, \ldots, n - 1$ sunt fixaţi implicit pe 0). Sumatorul cu transport anticipat consumă $O(\lg n)$ unităţi de timp. Astfel, întregul multiplicator a două numere de câte n biţi consumă $\Theta(\lg n)$ unităţi de timp.

[2] După cum se poate observa din figură, un arbore Wallace nu este cu adevărat un arbore, ci mai degrabă un graf aciclic orientat. Denumirea este istorică şi se păstrează încă în literatura de specialitate.

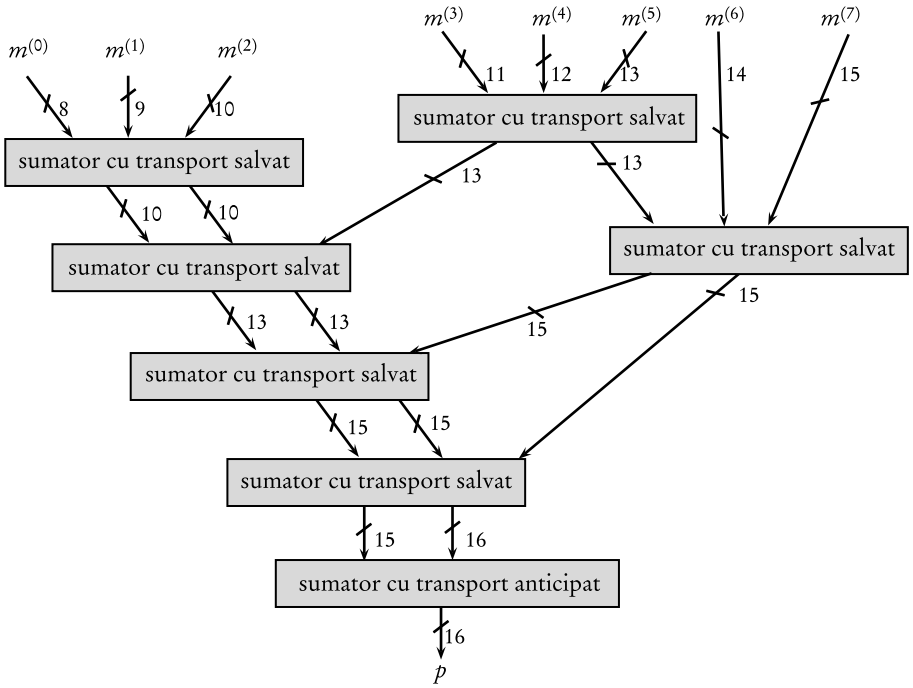


Figura 29.16 Un arbore Wallace care adună, pentru $n = 8$, produsele parţiale $m^{(0)}, m^{(1)}, \ldots, m^{(7)}$. Fiecare linie indică un număr având numărul de biţi indicat lângă linie. Ieşirea din stânga a fiecărui sumator cu transport salvat reprezintă biţii sumei, iar ieşirea din dreapta reprezintă transporturile.

Un multiplicator arbore Wallace pentru două numere de câte n biţi are dimensiunea $\Theta(n^2)$, după cum se va vedea imediat. Vom delimita, mai întâi, dimensiunile circuitelor sumatoare cu transport salvat. Se obţine uşor o margine inferioară de $\Omega(n^2)$, deoarece există $\lfloor n/3 \rfloor$ sumatoare cu transport salvat de adâncime 1 şi fiecare constă din cel puţin n sumatoare complete. Pentru a se obţine marginea superioară de $O(n^2)$, se va observa că produsul final are $2n$ biţi, fiecare sumator cu transport salvat din arborele Wallace conţine cel mult $2n$ sumatoare complete. Este necesar să arătăm că, de fapt, există numai $O(n)$ sumatoare complete. Fie $C(n)$ numărul de sumatoare cu salvare transport având n numere la intrare. Are loc formula de recurenţă:

$$C(n) \leq \begin{cases} 1 & \text{dacă } n = 3, \\ C(\lceil 2n/3 \rceil) + \lfloor n/3 \rfloor & \text{dacă } n \geq 4, \end{cases}$$

care are ca soluţie $C(n) = \Theta(n)$, dacă se consideră cazul 3 al teoremei master. Se obţine, astfel, o margine asimptotic strânsă de dimensiune $\Theta(n^2)$ pentru sumatoarele cu transport salvat din multiplicatorul arbore Wallace. Circuitele necesare pentru cele n produse parţiale au dimensiunea $\Theta(n^2)$, iar sumatorul cu transport anticipat de la sfârşit are dimensiunea $\Theta(n)$. Astfel, dimensiunea multiplicatorului este $\Theta(n^2)$.

Astfel, multiplicatorul bazat pe arborele Wallace este asimptotic mai rapid decât multiplicatorul matriceal. Deşi au aceeaşi dimensiune asimptotică, aspectul de implementare al arborelui Wallace nu este la fel de obişnuit ca cel al multiplicatorului matriceal, nici aşa de "dens" (în

sensul că are spaţii mai largi între elemente). În practică, se utilizează, de multe ori, o soluţie de compromis. Ideea este aceea că se folosesc, în paralel, două sumatoare matriceale: unul adună jumătate din produsele parţiale, iar celălalt adună cealaltă jumătate. Întârzierea de propagare este numai jumătate din cea produsă de un singur tablou care adună toate cele n produse parţiale. Două sumatoare cu transport salvat reduc ieşirea de 4 numere produse de tablouri la 2 numere, şi un sumator cu transport anticipat adună apoi cele două numere pentru a calcula produsul. Coada totală de propagare este mai mică decât jumătate din cea a unui multiplicator matriceal complet, plus un termen suplimentar de $O(\lg n)$.

Exerciţii

29.3-1 Arătaţi că, într-un multiplicator matriceal, avem $v_j^{(i)} = 0$, pentru $i = 1, 2, \ldots, n-1$ şi $j = 0, 1, \ldots, i, i+n, i+n+1, \ldots, 2n-1$.

29.3-2 Arătaţi că, în multiplicatorul matriceal din figura 29.14, sumatoarele complete din linia de sus, cu excepţia unuia, nu sunt toate necesare. Pentru aceasta, se impun o serie de schimbări în conexiuni.

29.3-3 Presupunem că un sumator cu transport salvat are intrările x, y şi z şi produce ieşirile s şi c având, respectiv, câte n_x, n_y, n_z, n_s şi n_c biţi. De asemenea, fără a pierde din generalitate, presupunem că $n_x \le n_y \le n_z$. Arătaţi atunci că $n_s = n_z$ şi că

$$n_c = \begin{cases} n_z & \text{dacă } n_y < n_z, \\ n_z + 1 & \text{dacă } n_y = n_z. \end{cases}$$

29.3-4 Arătaţi că înmulţirea poate fi realizată în timp $O(\lg n)$ şi dimensiunea $O(n^2)$ numai dacă impunem restricţia ca porţile să aibă evantai de intrare $O(1)$.

29.3-5 Descrieţi un circuit eficient care calculează câtul împărţirii la 3 a unui număr binar x divizibil cu 3. (*Indicaţie:* În binar, $0.010101\ldots = 0.01 \times 1.01 \times 1.0001 \times \cdots$.

29.3-6 Un ***deplasator ciclic*** este un circuit care are două intrări $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0 \rangle$ şi $s = \langle s_{m-1}, s_{m-2}, \ldots, s_0 \rangle$, unde $m = \lceil \lg n \rceil$. Ieşirea lui $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0 \rangle$ este $y_i = x_{i+s \bmod n}$, pentru $i = 0, 1, \ldots, n-1$. Altfel spus, acest deplasator roteşte biţii lui x cu s poziţii. Descrieţi un deplasator ciclic eficient. În termenii înmulţirii modulo, ce funcţie implementează un deplasator ciclic?

29.4. Circuite cu ceas

Elementele de circuite combinaţionale pot fi folosite de mai multe ori în timpul calculelor. Prin introducerea elementelor de ceas în circuite, aceste elemente combinaţionale pot fi reutilizate. De cele mai multe ori, circuitele cu ceas sunt mai mici decât circuitele combinaţionale construite pentru aceeaşi funcţie.

Această secţiune investighează circuitele cu ceas folosite pentru adunare şi înmulţire. Începem cu un circuit cu ceas de dimensiune $\Theta(1)$, numit sumator bit-serial, care adună două numere de câte n biţi în timp $\Theta(n)$. Apoi vom studia multiplicatorii matriceali liniari. Vom prezenta un multiplicator matriceal liniar de dimensiune $\Theta(n)$ care poate înmulţi două numere de câte n biţi într-un timp $\Theta(n)$.

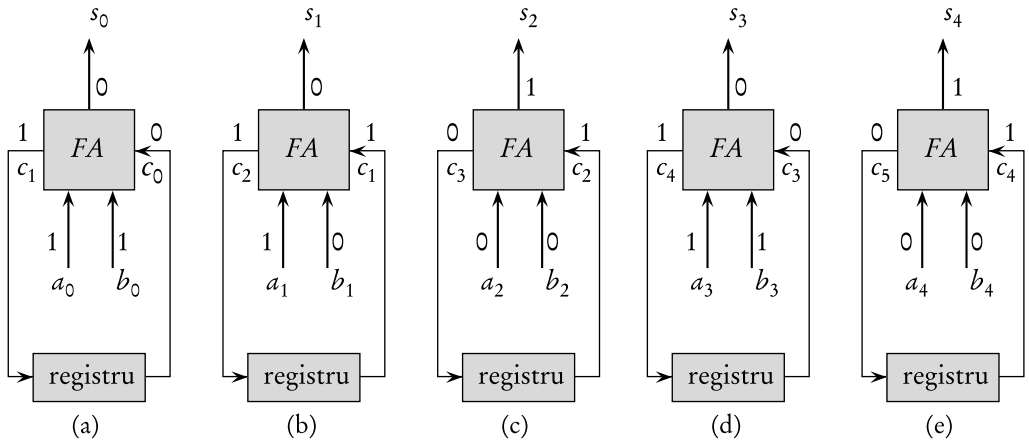


Figura 29.17 Operarea într-un sumator bit-serial. În timpul celui de-al i-lea ciclu de ceas, $i = 0, 1, \ldots, n$, sumatorul complet FA primeşte intrările a_i şi b_i, iar bitul de transport c_i din registru. Sumatorul complet produce bitul sumă s_i, iar bitul de transport c_{i+1}, este memorat înapoi în registru, spre a fi utilizat în tactul următor. Registrul este iniţializat cu $c_0 = 0$. **(a)-(e)** indică stările din cele 5 faze pentru adunarea numerelor $a = \langle 1011 \rangle$ şi $b = \langle 1001 \rangle$, rezultatul fiind $s = \langle 10100 \rangle$.

29.4.1. Sumator bit-serial

Introducem circuitul cu ceas revenind la problema adunării a două numere de câte n biţi. În figura 29.17 se arată cum se poate utiliza un singur sumator complet pentru a produce o sumă de $n + 1$ biţi $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$ a două numere de câte n biţi $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ şi $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Mediul extern prezintă perechile de biţi de intrare la fiecare tact: întâi, a_0 şi b_0, apoi, a_1 şi b_1 ş.a.m.d. Deşi se doreşte transportul unui bit la poziţia următoare, nu se pot furniza toate transporturile deodată. Este o chestiune de timp: transportul de intrare la poziţia c_i trebuie să intre în sumatorul complet la momentul în care sunt însumate intrările a_i, b_i. Absenţa acestei intrări, la timpul potrivit, conduce, evident, la un rezultat eronat.

Aşa cum rezultă din figura 29.17, soluţia este de a folosi un ***sumator cu circuit de ceas*** sau ***circuit secvenţial***, care conţine circuite combinaţionale şi unul sau mai mulţi ***regiştri*** (elemente de memorare cu ceas). Circuitele combinaţionale au intrări fie din mediul extern, fie din regiştrii de ieşire. Ele furnizează ieşiri fie spre mediul extern, fie în regiştri. Ca şi la circuitele combinaţionale, se interzice includerea, în circuitele cu ceas, a circuitelor combinaţionale care conţin cicluri.

Fiecare registru dintr-un circuit cu ceas este controlat printr-un semnal periodic, numit ***tact de ceas***. De fiecare dată când ceasul pulsează, sau dă un ***tact***, registrul încarcă şi memorează în el noua intrare. Intervalul de timp dintre două semnale (tacte de ceas) constituie ***perioada de ceas*** (***frecvenţa de ceas***). Într-un ***circuit global de ceas***, toţi regiştrii sunt coordonaţi de acelaşi ceas.

Să examinăm, puţin mai în detaliu, operaţiile cu un registru. Privim fiecare tact ca un puls momentan. La un anumit tact, registrul citeşte valoarea de intrare prezentată *în acel moment* şi o memorează. Această valoare memorată apare ca ieşire a registrului, atunci când este mutată într-un alt registru la următorul tact. Cu alte cuvinte, valoarea de intrare dintr-un registru la

un moment dat apare la ieşire la următorul tact de ceas.

Acum să examinăm circuitul din figura 29.17, pe care îl numim ***sumator bit-serial***. Pentru ca ieşirile sumatorilor să fie corecte, se cere ca intervalul dintre tacte să fie cel puţin tot atât de lung ca lungimea cozii de propagare, astfel încât să se poată efectua calculul la fiecare etapă în parte. În timpul perioadei 0, aşa cum se vede în figura 29.17(a), mediul extern aplică intrările a_0 şi b_0 la două dintre intrările sumatorului. Presupunem că, în registru, se află valoarea 0. Transportul iniţial este, astfel, $c_0 = 0$. Mai târziu, în acest tact se generează ieşirea s_0 şi noul transport c_1. Bitul de ieşire este preluat în mediul extern unde va fi salvat ca parte a sumei s. Firul transportorului sumatorului complet alimentează registrul, astfel încât c_1 este citit în registru la următorul tact de ceas. La începutul perioadei 1, (figura 29.17(b)) sosesc din exterior a_1 şi b_1, se citeşte transportul c_1 din registru, după care se produce ieşirea s_1 şi noul transport c_2, care se depune în registru. Bitul sumă s_1 este neglijat, iar bitul c_2 intră în registru. Acest ciclu continuă până la momentul n ilustrat în figura 29.17(e), în care registrul conţine c_n. Mediul extern transmite $a_n = b_n = 0$ şi, astfel, $s_n = c_n$.

Analiză

Pentru a determina timpul total t cerut de un circuit global de ceas, este necesar să se cunoască numărul de perioade de timp p şi durata d a fiecărei perioade, deci avem $t = pd$. Este necesar ca perioada d să fie suficient de mare pentru a se permite execuţia fiecărui tact. Desigur, în funcţie de intrări, calculul se poate desfăşura mai repede, însă d trebuie să fie proporţional cu adâncimea circuitului combinaţional.

Să vedem, acum, cât durează sumatorul bit-serial pentru numere de n biţi. Fiecare tact de ceas are timpul $\Theta(1)$ deoarece adâncimea este $\Theta(1)$. Deoarece se cere producerea a $n + 1$ biţi, timpul total se adună, deci $(n + 1)\Theta(1) = \Theta(n)$.

Dimensiunea sumatorului bit-serial (numărul de elemente combinaţionale plus numărul de regiştri) este $\Theta(1)$.

Adunarea cu transport propagat în comparaţie cu adunarea cu bit serial

Se observă că sumatorul cu transport propagat apare ca o dublare a sumatorului bit-serial cu regiştrii înlocuiţi cu conexiuni directe la elementele combinaţionale corespunzătoare. Deci sumatorul cu transport propagat este, de fapt, un fel de reprezentare spaţială a sumatorului bit-serial. Cel de-al i-lea sumator complet corespunde tactului i al execuţiei sumatorului bit-serial.

În general, orice circuit cu ceas poate fi înlocuit cu unul combinaţional echivalent având întârzierea de timp asimptotic egală cu primul, cu condiţia să cunoaştem dinainte numărul de tacte executate de circuitul cu ceas. Există, evident, o serie de diferenţe. Circuitul cu ceas foloseşte mai puţine elemente combinaţionale (mai puţine cu un factor de $\Theta(n)$ în cazul unui sumator bit serial comparat cu un sumator cu transport propagat.) În schimb, circuitul combinaţional are avantajul unei părţi de control mai puţin complicate – nu este nevoie de un circuit cu ceas sau un circuit extern sincronizat pentru a transmite biţi de intrare sau pentru a memora biţi de sumă. Mai mult, circuitele combinaţionale sunt puţin mai rapide decât cele cu ceas, deci mai eficiente în practică. Câştigul de timp provine din faptul că nu se aşteaptă după stabilizare în fiecare tact.

a	b
19	29
9	58
4	116
2	232
1	464
	551

(a)

a	b
10011	11101
1001	111010
100	1110100
10	11101000
1	111010000
	1000100111

(b)

Figura 29.18 Înmulţirea lui 19 cu 29 folosind algoritmul ţăranului rus. O linie din coloana lui a se obţine prin înjumătăţirea, prin lipsă, a numărului din linia precedentă, iar o intrare din coloana lui b se obţine prin dublarea numărului din linia precedentă. Adunăm elementele din liniile corespunzătoare ale coloanei b cu elemente având pe linia respectivă valori impare din coloana a (haşurate). Suma obţinută este produsul căutat. **(a)** Numerele scrise în zecimal. **(b)** Aceleaşi numere în binar.

29.4.2. Multiplicatori liniari

Multiplicatorii combinaţionali din secţiunea 29.3 necesită dimensiuni de $\Theta(n^2)$ pentru a înmulţi două numere de câte n biţi. Vom prezenta doi algoritmi în care circuitele se pot reprezenta liniar, nu sub forma unui tablou bidimensional de circuite. Ca şi la multiplicatorul matriceal, timpul de execuţie este $\Theta(n)$.

Multiplicatorii liniari implementează ***algoritmul ţăranului rus*** (numit aşa pentru faptul că nişte vestici care au vizitat Rusia în secolul 19, au găsit acest algoritm foarte răspândit aici). Algoritmul este ilustrat în figura 29.18(a). Fiind date două numere a şi b, se formează două coloane de numere având, în antetul tabelului, numerele a şi b. În fiecare linie din coloana a se trece jumătatea, prin lipsă, a numărului din linia precedentă. În fiecare linie din coloana b se trece dublul numărului din linia precedentă. Completarea tabelului se face linie cu linie până când în coloana a se obţine valoarea 1. În acest moment, sunt însumate liniile din b care au în a valori impare, obţinându-se valoarea produsului $a \cdot b$.

Algoritmul ţăranului rus apare, la prima vedere, cel puţin ciudat, dacă nu remarcabil. Dacă însă urmărim figura 29.18(b), se constată că este vorba de înmulţirea clasică scrisă în baza 2 (deînmulţitul deplasat se adună când cifra curentă a înmulţitorului este 1, adică atunci când (după deplasare) este număr impar.)

O implementare lentă a multiplicatorului liniar

În figura 29.19(a) este prezentată o implementare a algoritmului ţăranului rus pentru două numere de câte n biţi. Se foloseşte un circuit cu ceas constând dintr-un tablou liniar de $2n$ celule. Fiecare celulă conţine trei regiştri. Unul dintre regiştri conţine un bit din intrarea a, alt registru conţine un bit din intrarea b, iar al treilea un bit al produsului p. Indicii superiori notează valorile din celule înaintea fiecărui pas al algoritmului. De exemplu, valoarea bitului a_i înaintea pasului j este $a_i^{(j)}$ şi defineşte numărul $a^{(j)} = \langle a_{2n-1}^{(j)}, a_{2n-2}^{(j)}, \ldots, a_0^{(j)} \rangle$.

Algoritmul execută o secvenţă de n paşi, numerotaţi $0, 1, \ldots, n-1$, unde fiecare pas este

executat într-un tact. Algoritmul păstrează invariantul, indiferent de pasul j:

$$a^{(j)} \cdot b^{(j)} + p^{(j)} = a \cdot b \tag{29.6}$$

(vezi exerciţiul 29.4-2). Iniţial, $a^{(0)} = a$, $b^{(0)} = b$ şi $p^{(0)} = 0$. Pasul j constă din următoarele calcule:

1. Dacă $a^{(j)}$ este impar (adică $a_0^{(j)} = 1$), atunci adună b la p: $p^{(j+1)} \leftarrow b^{(j)} + p^{(j)}$. (Adunarea este efectuată de un sumator cu transport propagat care determină lungimea tabloului; bitul de transport se propagă de la dreapta la stânga.) Dacă $a^{(j)}$ este par, atunci p este modificat la pasul următor: $p^{(j+1)} \leftarrow p^{(j)}$.

2. Deplasează a spre dreapta cu o poziţie:

$$a_i^{(j+1)} \leftarrow \begin{cases} a_{i+1}^{(j)} & \text{dacă } 0 \le i \le 2n-2, \\ 0 & \text{dacă } i = 2n-1. \end{cases}$$

3. Deplasează b spre stânga cu o poziţie:

$$b_i^{(j+1)} \leftarrow \begin{cases} b_{i-1}^{(j)} & \text{dacă } 1 \le i \le 2n-1, \\ 0 & \text{dacă } i = 0. \end{cases}$$

După execuţia celor n paşi, s-au calculat toţi biţii lui a; atunci $a^{(n)} = 0$. Invariantul (29.6) implică $p^{(n)} = a \cdot b$.

Să analizăm acest algoritm. Presupunând că informaţiile se propagă în toate celulele simultan, avem n paşi. Fiecare pas consumă, în cel mai defavorabil caz, un timp de $\Theta(n)$ deoarece adâncimea sumatorului cu transport propagat este $\Theta(n)$. O perioadă de ceas este $\Theta(n)$. Fiecare deplasare se execută în $\Theta(1)$ unităţi de timp. Din aceste considerente, algoritmul consumă un timp $\Theta(n^2)$. Deoarece fiecare celulă are dimensiune constantă, dimensiunea multiplicatorului liniar este $\Theta(n)$.

O implementare liniară rapidă

Folosind sumatori cu transport salvat în loc de sumatori cu transport propagat, timpul fiecărui pas scade la $\Theta(1)$, obţinând astfel să scadă timpul total la $\Theta(n)$. Aşa cum reiese din figura 29.19(b), o dată cu fiecare intrare a unui bit din intrarea a, intră un bit şi în intrarea b. Fiecare celulă conţine doi biţi în plus, u, v, care sunt ieşirile de salvare a transporturilor. Folosind o reprezentare pentru transport salvat la însumările produselor, invariantul se păstrează la fiecare pas j,

$$a^{(j)} \cdot b^{(j)} + u^{(j)} + v^{(j)} = a \cdot b \tag{29.7}$$

(vezi exerciţiul 29.4-2). Fiecare pas deplasează a şi b la fel ca la algoritmul liniar lent, astfel încât putem combina ecuaţiile (29.6) şi (29.7) de unde rezultă că $u^{(j)} + v^{(j)} = p^{(j)}$. Aşadar, biţii u şi v conţin aceeaşi informaţie ca şi p în algoritmul lent.

Al j-lea pas din implementare efectuează adunarea cu salvare transport pentru u şi v, unde operanzii depind de faptul că a este par sau nu. Dacă $a_0^{(j)} = 1$, atunci se calculează:

$$u_i^{(j+1)} \leftarrow \text{ paritate}(b_i^{(j)}, u_i^{(j)}, v_i^{(j)}), \text{ pentru } i = 0, 1, \ldots, 2n-1$$

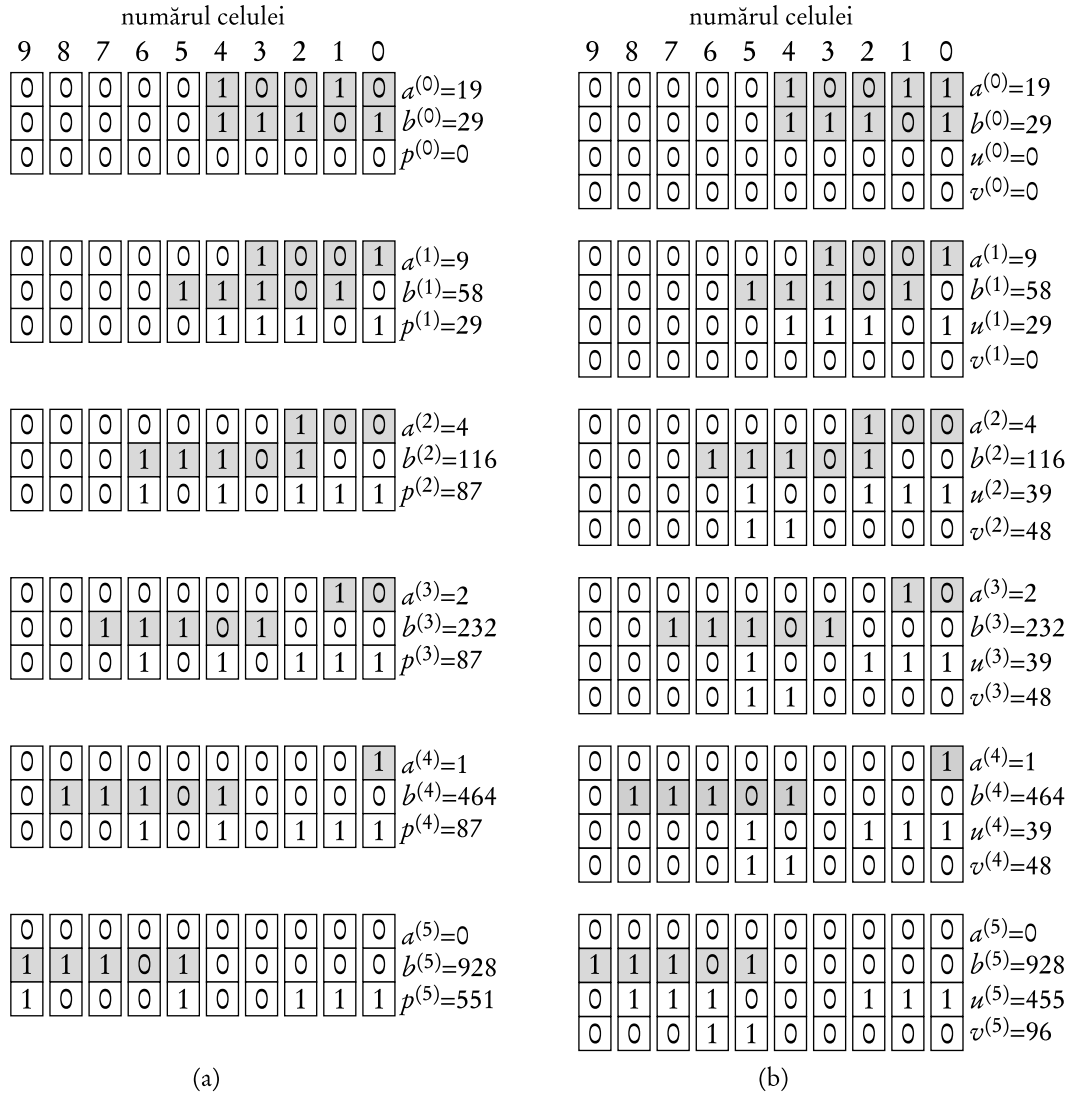


Figura 29.19 Doi multiplicatori liniari care implementează algoritmul ţăranului rus, ambii înmulţesc $a = 19 = \langle 10011 \rangle$ cu $b = 29 = \langle 11101 \rangle$ şi $n = 5$. Este prezentată situaţia la începutul fiecărui pas j, iar biţii semnificativi din $a^{(j)}b^{(j)}$ ai pasului sunt haşuraţi. **(a)** Implementarea lentă necesită un timp $\Theta(n^2)$. Deoarece $a^{(5)} = 0$, avem $p^{(5)} = a \cdot b$. Sunt efectuaţi n paşi. În fiecare, este aplicat un sumator cu transport propagat. Tactul este proporţional cu lungimea tabloului, respectiv $\Theta(n)$, conducând la un timp total de $\Theta(n^2)$. (**(b)** Implementarea rapidă necesită doar timp $\Theta(n)$, deoarece prin folosirea sumatorului cu transport salvat o însumare necesită doar un timp $\Theta(1)$. Sunt necesari, în total, $2n - 1 = 9$ paşi. După prezentarea ultimului pas, adunările repetate ale transporturilor din u şi v produc $u^{(9)} = a \cdot b$.

şi

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{majoritate}(b_{i-1}^{(j)}, u_{i-1}^{(j)}, v_{i-1}^{(j)}) & \text{dacă } 1 \le i \le 2n-1, \\ 0 & \text{dacă } i = 0. \end{cases}$$

În caz contrar, dacă $a_0^{(j)} = 0$, calculăm

$$u_i^{(j+1)} \leftarrow \text{ paritate}(0, u_i^{(j)}, v_i^{(j)}), \text{ pentru } i = 0, 1, \ldots, 2n-1$$

şi

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{majoritate}(0, u_{i-1}^{(j)}, v_{i-1}^{(j)}) & \text{dacă } 1 \le i \le 2n-1, \\ 0 & \text{dacă } i = 0. \end{cases}$$

După actualizarea lui u şi a lui v, în pasul j, se deplasează a la dreapta şi b la stânga în aceeaşi manieră ca la algoritmul lent.

Implementarea rapidă efectuează, în total, $2n-1$ paşi. Pentru $j \ge n$, avem $a^{(j)} = 0$ şi invariantul (29.7) implică $u^{(j)} + v^{(j)} = a \cdot b$. O dată ce $a^{(j)} = 0$, toţi paşii viitori servesc numai pentru a aduna transporturile salvate la u şi v. Exerciţiul 29.4-3 arată că $v^{(2n-1)} = 0$, deci $u^{(2n-1)} = a \cdot b$.

Timpul total, în cel mai defavorabil caz, este $\Theta(n)$ deoarece fiecare din cei $2n-1$ paşi consumă un timp $\Theta(1)$. Deoarece fiecare celulă consumă timp constant, timpul total rămâne $\Theta(n)$.

Exerciţii

29.4-1 Fie $a = \langle 101101 \rangle$, $b = \langle 011110 \rangle$ şi $n = 6$. Arătaţi, pentru a, b daţi, cum operează algoritmul ţăranului rus, atât în binar, cât şi în zecimal.

29.4-2 Demonstraţi valabilitatea invarianţilor (29.6) şi (29.7) pentru multiplicatorii liniari.

29.4-3 Arătaţi că $v^{(2n-1)} = 0$ în multiplicatorul liniar rapid.

29.4-4 Descrieţi cum reprezintă multiplicatorul matriceal din secţiunea 29.3.1 calculul "nestratificat" al multiplicatorului liniar rapid.

29.4-5 Considerăm un flux de date $\langle x_1, x_2, \ldots \rangle$ care soseşte la un circuit cu ceas cu o rată de o valoare pe tact. Pentru o valoare fixată n, circuitul trebuie să calculeze valoarea:

$$y_t = \max_{t-n+1 \le i \le t} x_i$$

pentru $t = n, n+1, \ldots$. Astfel, y_t este valoare maximă dintre cele mai recente n valori sosite la circuit. Daţi un circuit cu dimensiunea $O(n)$ care, în fiecare tact cu intrarea x_t, calculează valoarea de ieşire y_t în timp $O(1)$. Circuitul poate utiliza regiştri şi elemente combinaţionale care calculează maximul dintre două intrări.

29.4-6 $\star$ Rezolvaţi din nou exerciţiul 29.4-5 folosind numai $O(\lg n)$ elemente de tip "maximum".

Probleme

29-1 Circuite de împărţire

Pornind de la tehnica numită ***iteraţie Newton***, se poate construi un circuit de împărţire folosind circuite de scădere şi de înmulţire. Problema se rezolvă plecând de la problema înrudită de calculare a inversului, obţinând astfel un circuit de împărţire pornind de la un circuit adiţional de înmulţire.

Ideea este de a calcula o secvenţă $y_0, y_1, y_2, \ldots$ de aproximare a inversului unui număr x folosind formula:

$$y_{i+1} \leftarrow 2y_i - xy_i^2.$$

Presupunem că este dat un număr binar fracţionar x având n cifre după virgulă şi $1/2 \leq x \leq 1$. Deoarece inversul poate avea o infinitate de zecimale, vom calcula cele mai semnificative n cifre ale lui.

a. Presupunem că $|y_i - 1/x| \leq \epsilon$, pentru un $\epsilon > 0$ fixat. Arătaţi că $|y_{i+1} - 1/x| \leq \epsilon^2$.

b. Daţi o aproximaţie y_0, astfel încât y_k să satisfacă relaţia $|y_k - 1/x| \leq 2^{-2^k}$, pentru orice $k \geq 0$. Cât de mare trebuie să fie k astfel încât aproximaţia y_k să fie potrivită, inclusiv pentru cea mai puţin semnificativă dintre cele n zecimale?

c. Descrieţi un circuit combinaţional care, dacă i se dă o intrare x de n biţi, aproximează $1/x$ în timp $O(\lg^2 n)$. Care este dimensiunea acestui circuit? (*Indicaţie:* Cu puţină isteţime, se poate învinge limitarea dimensiunii la $\Theta(n^2 \lg n)$.)

29-2 Formule booleene pentru funcţii simetrice

O funcţie de n variabile $f(x_1, x_2, \ldots, x_n)$ este ***simetrică*** dacă

$$f(x_1, x_2, \ldots, x_n) = f(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$$

pentru fiecare permutare π a mulţimii $\{1, 2, \ldots, n\}$. În această problemă presupunem că există o formulă booleană ce reprezintă pe f şi a cărei lungime este polinomială în n. (Pentru scopul nostru, o formulă booleană este un şir care cuprinde variabilele $x_1, x_2, \ldots, x_n$, paranteze şi operatorii booleeni $\wedge$, $\vee$ şi $\neg$.) Ideea de rezolvare este de a converti un circuit boolean de adâncime logaritmică într-o formulă booleană de lungime polinomială. Presupunem că sunt folosite câte două circuite poartă AND, OR, NOT.

a. Începem construind o funcţie simetrică simplă: ***funcţia majoritate*** generalizată pe n intrări boolene, definită prin:

$$\text{majoritate}_n(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{dacă } x_1 + x_2 + \ldots + x_n > n/2, \\ 0 & \text{altfel.} \end{cases}$$

Descrieţi un circuit de adâncime $O(\lg n)$ pentru funcţia majoritate_n. (*Indicaţie:* Construiţi un arbore de sumatori.)

b. Presupunem că f este o funcţie booleană oarecare de n variabile $x_1, x_2, \ldots, x_n$. Pesupunem că există un circuit C de adâncime d care îl calculează pe f. Arătaţi cum se construieşte, pentru C, o formulă booleană de lungime $O(2^d)$. Deduceţi că există pentru majoritate$_n$ o formulă de dimensiune polinomială.

c. Arătaţi că orice funcţie booleană simetrică $f(x_1, x_2, \ldots, x_n)$ poate fi exprimată ca o funcţie de $\sum_{i=1}^{n} x_i$.

d. Arătaţi că orice funcţie simetrică de n intrări booleene poate fi calculată de către un circuit combinaţional de adâncime $O(\lg n)$.

e. Arătaţi că orice funcţie booleană de n variabile booleene poate fi reprezentată printr-o formulă de dimensiune polinomială în n.

Note bibliografice

Cele mai multe cărţi de aritmetică şi calculatoare se concentrează mai mult pe aspectele practice ale implementării circuitelor decât pe o teorie algoritmică. Savage [173] este unul dintre puţinii care a cercetat aspecte algoritmice în acest domeniu. Recomandăm cărţile de aritmetica calculatoarelor, orientate hardware, ale lui Cavanagh [39] şi Hwang [108]. În ceea ce priveşte proiectarea circuitelor logice, combinaţionale şi secvenţiale, recomandăm cartea lui Hill şi Peterson [96], şi, în combinaţie cu teoria limbajelor formale, cea a lui Kohavi [126].

Primele urme istorice ale algoritmicii aritmetice le întâlnim la Aiken şi Hopper [7]. Algoritmii de însumare cu transport propagat sunt tot aşa de vechi ca şi abacul, care are cel puţin 5000 de ani. Primul calculator mecanic care a implementat acest tip de sumator este cel al lui B. Pascal în 1642. S. Morland în 1666 şi, independent de el, G. W. Leibnitz în 1671 au creat maşini de multiplicat prin adunări succesive. Înmulţirea folosind algoritmul ţăranului rus se pare că este mai veche decât folosirea ei în Rusia secolului 19. După Knuth [122], metoda a fost folosită de matematicienii egipteni încă înainte de 1800 Î.Hr.

Utilizarea stării şirurilor de transport: distrugere, generare, propagare, a fost aplicată la primul calculator construit la Harvard în jurul anului 1940 [180]. Una dintre primele implementări ale sumatorului cu transport anticipat este descrisă de Weinberger şi Smith [199], dar metoda lor de anticipare cere porţi foarte mari. Ofman [152] demonstrează că numerele de n biţi pot fi însumate, folosind un sumator cu transport anticipat, în timp $O(\lg n)$ şi folosind porţi de dimensiune constantă.

Ideea folosirii adunării cu transport salvat pentru accelerarea înmulţirii, aparţine autorilor Estrin, Gilchrist şi Pomerene [64]. Atrubin [13] descrie un multiplicator liniar de lungime infinită ce poate fi folosit pentru înmulţirea numerelor oricât de mari. Multiplacatorul produce cel de-al n-lea bit al produsului imediat ce primeşte al n-lea bit al intrării. Multiplicatorul arbore Wallace este atribuit lui Wallace [197], însă ideea lui a fost descoperită independent şi de Ofman [152].

Algoritmii de împărţire sunt datoraţi lui Newton, care, încă în jurul anului 1665, a inventat ceea ce se cheamă metoda iterativă a lui Newton. Problema 29-1 foloseşte această metodă pentru a construi un circuit de împărţire cu adâncimea $\Theta(\lg^2 n)$. Justificarea acestei metode a fost dată de Beame, Cook şi Hoover [19], care au arătat că împărţirea pe n biţi poate fi efectuată, de fapt, cu adâncimea de $\Theta(\lg n)$.

30 Algoritmi pentru calculatoare paralele

Datorită creşterii numărului calculatoarelor cu procesare paralelă, a crescut şi interesul pentru ***algoritmii paraleli***: algoritmi care efectuează mai multe operaţii în acelaşi timp. Studiul algoritmilor paraleli a devenit, la ora actuală, un nou domeniu de cercetare, de sine stătător. Într-adevăr, au fost dezvoltaţi algoritmi paraleli pentru multe probleme care au fost rezolvate folosind algoritmi seriali obişnuiţi. În acest capitol vom descrie câţiva algoritmi paraleli simpli care ilustrează probleme şi tehnici fundamentale.

Pentru a studia algoritmii paraleli, trebuie să alegem un model adecvat pentru calculul paralel. Maşina cu acces aleator, sau RAM (engl. Random-Acces Machine), folosită în cea mai mare parte a acestei lucrări, este o maşină serială şi nu una paralelă. Modelele paralele pe care le-am studiat – reţele de sortare (capitolul 28) şi circuitele (capitolul 29) – sunt prea restrictive pentru a cerceta, de exemplu, algoritmi pe structuri de date.

Algoritmii paraleli, din acest capitol, sunt prezentaţi în termenii unui model teoretic cunoscut: maşina paralelă cu acces aleator (engl. Parallel Random-Acces Machine), sau PRAM (se pronunţă "PE-ram"). Mulţi algoritmi paraleli pentru şiruri, liste, arbori sau grafuri pot fi foarte uşor descrişi folosind modelul PRAM. Cu toate că acest model ignoră multe aspecte importante ale maşinilor paralele reale, atributele esenţiale ale algoritmilor paraleli tind să nu fie limitaţi de modelele pentru care au fost proiectaţi. Dacă un algoritm PRAM este mai bun decât un alt algoritm PRAM, este puţin probabil ca performanţele relative ale acestor algoritmi să se schimbe în mod substanţial când ambii algoritmi sunt adaptaţi execuţiei pe un calculator real cu procesare paralelă.

Modelul PRAM

În figura 30.1 este prezentată arhitectura de bază a unei ***maşini paralele cu acces aleator (PRAM)***. Există p procesoare (seriale) obişnuite $P_0, P_1, \ldots, P_{p-1}$ care au ca spaţiu de stocare o memorie globală partajată. Toate aceste procesoare pot citi sau scrie în această memorie "în paralel" (în acelaşi timp). De asemenea, procesoarele pot efectua diferite operaţii aritmetice şi logice în paralel.

Ipoteza de bază în privinţa performanţelor algoritmilor PRAM este aceea că timpul de execuţie poate fi măsurat prin numărul accesărilor paralele la memorie efectuate de algoritm. Această ipoteză reprezintă o generalizare directă a modelelor RAM obişnuite, în care numărul accesărilor este, asimptotic, la fel de bun ca orice altă modalitate de a măsura timpul de execuţie. Această ipoteză simplă ne va fi foarte folositoare în cercetarea algoritmilor paraleli, chiar şi atunci când calculatoarele reale cu procesare paralelă nu pot executa accesări paralele la memoria globală într-o unitate de timp: timpul necesar unei accesări creşte o dată cu numărul procesoarelor din calculator.

Cu toate acestea, pentru algoritmii paraleli care accesează datele într-un anume fel, poate fi justificată ipoteza efectuării operaţiilor cu memoria în unitatea de timp. Maşinile paralele reale au, de obicei, o reţea de comunicaţie care poate simula această memorie globală. Accesarea datelor prin intermediul acestei reţele este o operaţie relativ lentă în comparaţie cu alte operaţii cum ar fi cele aritmetice. De aceea, numărând accesările paralele la memorie efectuate de doi algoritmi paraleli, obţinem, de fapt, o estimare destul de exactă a performanţelor lor relative.

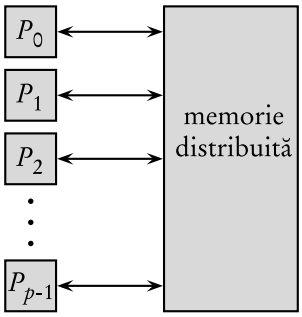


Figura 30.1 Arhitectura de bază a maşinii paralele cu acces aleator. Există p procesoare P_0, P_1, ..., P_{p-1} conectate la o memorie partajată. Fiecare procesor poate accesa un cuvânt arbitrar al memoriei partajate în unitatea de timp.

Motivul principal pentru care maşinile reale încalcă această abstractizare a unităţii de timp, este acela că anumite modele de accesare a memoriei sunt mai rapide decât altele. Totuşi, ca o primă aproximare, ipoteza unităţii de timp pentru modelele PRAM este destul de rezonabilă.

Timpul de execuţie al unui algoritm paralel depinde atât de numărul procesoarelor care execută acel algoritm cât şi de dimensiunea datelor de intrare. De aceea, în general, va trebui să luăm în considerare atât timpul cât şi numărul de procesoare când analizăm algoritmii PRAM; aceasta este o mare deosebire faţă de algoritmii seriali a căror analiză se concentra mai ales asupra timpului. De obicei, există o legătură între numărul de procesoare folosite de un algoritm şi timpul său de execuţie. Acest subiect va fi discutat în secţiunea 30.3.

Accesarea concurentă şi accesarea exclusivă

Un algoritm cu ***citire concurentă*** este un algoritm PRAM în timpul execuţiei căruia mai multe procesoare pot citi, în acelaşi timp, din aceeaşi locaţie a memoriei partajate. Un algoritm cu ***citire exclusivă*** este un algoritm PRAM în timpul execuţiei căruia două procesoare nu pot citi, din aceeaşi locaţie de memorie, în acelaşi timp. Facem o distincţie similară pentru algoritmii în cadrul execuţiei cărora două procesoare pot sau nu să scrie în aceeaşi locaţie de memorie în acelaşi timp, împărţind algoritmii PRAM în algoritmi cu ***scriere concurentă*** şi algoritmi cu ***scriere exclusivă***. Abrevierile folosite, de obicei, pentru tipurile de algoritmi pe care îi vom întâlni sunt:

- ***EREW***: citire exclusivă şi scriere exclusivă
- ***CREW***: citire concurentă şi scriere exclusivă
- ***ERCW***: citire exclusivă şi scriere concurentă
- ***CRCW***: citire concurentă şi scriere concurentă

Aceste prescurtări provin din limba engleză şi, de obicei, nu sunt pronunţate într-un singur cuvânt ci ca secvenţe de litere. Literele E, C, R şi W provin de la cuvintele *exclusive*, *concurrent*, *read*, *write* care se traduc prin *exclusiv*, *concurent*, *citire*, respectiv *scriere*.

Dintre aceste tipuri de algoritmi, extremele – EREW şi CRCW – sunt cele mai populare. O maşină care suportă numai algoritmi EREW este numită ***EREW PRAM***, iar una care suportă şi algoritmi CRCW poartă denumirea de ***CRCW PRAM***. Este evident că o maşină CRCW PRAM poate executa algoritmi EREW, dar o maşină EREW PRAM nu poate suporta, direct, accesările concurente ale memoriei necesare unui algoritm CRCW. Hardware-ul necesar unei maşini EREW PRAM este rapid deoarece nu este foarte complex, nefiind necesară tratarea unor conflicte cauzate de scrieri şi citiri simultane. O maşină CRCW PRAM necesită suport hardware suplimentar pentru ca ipoteza unităţii de timp să dea o măsură acceptabilă a performanţelor algoritmului. Totuşi, această maşină oferă un model de programare care este mai direct decât cel oferit de maşinile EREW PRAM.

Dintre celelalte două tipuri de algoritmi – CREW şi ERCW – literatura de specialitate a acordat mai multă atenţie tipului CREW. Totuşi, din punct de vedere practic, suportul pentru scrierea concurentă nu este mai dificil de realizat decât suportul pentru citirea concurentă. În general, în acest capitol vom trata algoritmii ca fiind de tip CRCW dacă vor conţine citiri sau scrieri concurente, fără a mai face alte distincţii. Vom discuta diferite aspecte ale acestei distincţii în secţiunea 30.2.

Când mai multe procesoare scriu în aceeaşi locaţie în timpul execuţiei unui algoritm CRCW, efectul scrierii paralele nu este bine definit fără clasificări suplimentare. În acest capitol, vom folosi modelul ***CRCW-comun***: când mai multe procesoare scriu în aceeaşi locaţie, ele trebuie să scrie aceeaşi valoare (comună). În literatura de specialitate, sunt prezentate mai multe tipuri alternative de maşini PRAM care tratează această problemă pornind de la alte ipoteze. Alte posibilităţi sunt:

- ***CRCW-arbitrar***: stochează o valoare arbitrară dintre cele care se încearcă a fi scrise
- ***CRCW-prioritar***: stochează valoarea pe care încearcă să o scrie procesorul cu indicele cel mai mic
- ***CRCW-combinat***: stochează o valoare obţinută printr-o combinaţie specificată a valorilor care se încearcă a fi scrise

În cazul ultimului tip, combinaţia specificată este, de obicei, o funcţie asociativă şi comutativă cum ar fi adunarea (se stochează suma valorilor care se încearcă a fi scrise) sau maximul (se stochează maximul valorilor care se încearcă a fi scrise).

Sincronizare şi control

Algoritmii PRAM trebuie să fie bine sincronizaţi pentru a lucra corect. Cum este obţinută această sincronizare? De asemenea, deseori, procesoarele care execută algoritmi PRAM trebuie să detecteze condiţii de terminare sau de ciclare care depind de starea tuturor procesoarelor. Cum este implementată această funcţie de control?

Nu vom discuta detaliat aceste probleme. Multe calculatoare reale cu procesare paralelă conţin o reţea de control care conectează procesoarele şi care ajută la condiţiile de terminare şi sincronizare. De obicei, reţeaua de control poate implementa aceste funcţii în acelaşi timp în care o reţea de rutaj poate implementa referinţe la memoria globală.

Pentru scopurile noastre, este suficient să presupunem că aceste procesoare sunt foarte bine sincronizate. La un moment dat, indicele instrucţiunii executate este acelaşi pentru toate

procesoarele. Nici un procesor nu avansează în timp ce altele execută instrucţiuni care se află înainte în cod. În timpul prezentării primului algoritm paralel, vom puncta momentele în care presupunem că procesoarele sunt sincronizate.

Pentru a detecta terminarea unei instrucţiuni paralele de ciclare care depinde de starea tuturor procesoarelor, vom presupune că o condiţie de terminare paralelă poate fi testată în reţeaua de control în timp $O(1)$. Unele modele EREW PRAM prezentate în literatura de specialitate nu iau în considerare această ipoteză şi timpul (logaritmic) necesar testării condiţiei de ciclare trebuie inclus în timpul total de execuţie (vezi exerciţiul 30.1-8). După cum vom vedea în secţiunea 30.2, maşinile CRCW PRAM nu au nevoie de o reţea de control pentru a testa terminarea: ele pot detecta terminarea unei instrucţiuni paralele de ciclare în timp $O(1)$ cu ajutorul scrierilor concurente.

Rezumatul capitolului

În secţiunea 30.1, vom introduce tehnica saltului de pointer care oferă o modalitate rapidă de a manipula liste în paralel. Vom arăta cum saltul de pointer poate fi folosit pentru a realiza calcule de prefixe pe liste şi cum unii algoritmi rapizi cu liste pot fi adaptaţi pentru lucrul cu arbori. În secţiunea 30.2, vom discuta puterea relativă a algoritmilor CRCW şi EREW şi vom arăta că accesarea concurentă a memoriei oferă o putere mai mare de execuţie.

În secţiunea 30.3, va fi prezentată teorema lui Brent care arată cum maşinile PRAM pot simula eficient circuitele combinaţionale. În această secţiune, vom discuta problema importantă a eficienţei muncii şi vom prezenta condiţiile care trebuie îndeplinite pentru ca un algoritm PRAM care foloseşte p procesoare (algoritm PRAM p-procesor) să poată fi transformat într-un algoritm PRAM p'-procesor pentru orice $p' < p$. În secţiunea 30.4, se va relua problema realizării calculelor de prefixe într-o listă înlănţuită şi va fi prezentat modul în care un algoritm probabilist poate efectua eficient calculul. În final, în secţiunea 30.5, se va arăta cum poate fi repartizată simetria în paralel, folosind un algoritm determinist. Timpul de execuţie va fi unul mult mai bun decât cel logaritmic.

Algoritmii paraleli prezentaţi în acest capitol au fost preluaţi, în special, din domeniul teoriei grafurilor. Ei reprezintă numai o mică selecţie a algoritmilor paraleli cunoscuţi în prezent. Totuşi, tehnicile introduse în acest capitol sunt destul de reprezentative pentru tehnicile folosite pentru algoritmii probabilişti din alte domenii ale informaticii.

30.1. Saltul de pointer

Printre cei mai importanţi algoritmi PRAM, se află cei care implică pointeri. În această secţiune, vom investiga o tehnică puternică numită salt de pointer care duce la algoritmi rapizi pentru operaţiile cu liste. Mai exact, vom introduce un algoritm care rulează în timp $O(\lg n)$ şi care determină distanţa faţă de capătul listei pentru fiecare obiect dintr-o listă cu n obiecte. Vom modifica, apoi, acest algoritm pentru a realiza un calcul paralel al "prefixului" pentru o listă cu n obiecte în timp $O(\lg n)$. În final, vom cerceta o metodă care permite ca mai multe probleme cu arbori să fie convertite în probleme cu liste care vor putea fi rezolvate cu ajutorul saltului de pointer. Toţi algoritmii din această secţiune sunt algoritmi EREW, nici o accesare concurentă la memoria globală nefiind necesară.

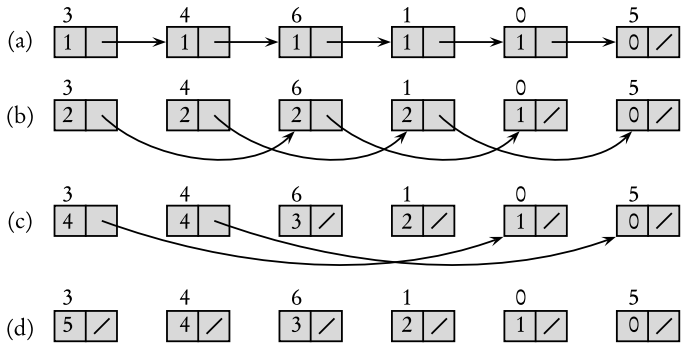


Figura 30.2 Determinarea distanţei de la fiecare obiect al unei liste cu n obiecte până la capătul listei în timp $O(\lg n)$ folosind saltul de pointer. **(a)** O listă înlănţuită reprezentată într-o maşină PRAM cu valorile d iniţializate. La sfârşitul algoritmului, fiecare valoare d va păstra distanţa de la obiectul corespunzător la capătul listei. Pentru fiecare obiect, procesorul responsabil de acel obiect apare deasupra. **(b)-(d)** Pointerii şi valorile d după fiecare iteraţie a buclei **cât timp** în algoritmul RANG-LISTĂ.

30.1.1. Determinarea rangului obiectelor unei liste

Primul nostru algoritm paralel operează cu liste. Vom putea stoca o listă într-o maşină PRAM la fel cum am stocat o listă într-o maşină RAM obişnuită. Totuşi, pentru a opera în paralel cu obiectele listei, este convenabil să atribuim fiecărui obiect un procesor "responsabil". Vom presupune că numărul obiectelor din listă nu depăşeşte numărul procesoarelor disponibile, şi că al i-lea procesor este "responsabil" de al i-lea obiect. De exemplu, în figura 30.2(a) este prezentată o listă înlănţuită constând dintr-un şir de obiecte $\langle 3, 4, 6, 1, 0, 5\rangle$. Deoarece există câte un procesor pentru fiecare obiect al listei, operaţiile efectuate de către un procesor asupra obiectului corespunzător pot fi realizate în timp $O(1)$.

Să presupunem că se dă o listă simplu înlănţuită L cu n obiecte şi dorim să determinăm, pentru fiecare obiect din L, distanţa până la capătul listei. Mai exact, dacă *urm* este câmpul pointerului, vom dori să calculăm o valoare $d[i]$ pentru fiecare obiect i al listei, astfel încât

$$d[i] = \begin{cases} 0 & \text{dacă } urm[i] = \text{NIL}, \\ d[urm[i]] + 1 & \text{dacă } urm[i] \neq \text{NIL}. \end{cases}$$

Vom numi problema determinării valorilor d ***problema determinării rangului elementelor unei liste***.

O soluţie a problemei determinării rangului elementelor unei liste este de a propaga, pur şi simplu, distanţa de la capătul listei. Această metodă necesită un timp $\Theta(n)$ deoarece al k-lea obiect din capătul listei trebuie să aştepte ca cele $k-1$ obiecte care îl urmează să-şi determine distanţele până la capăt înainte de a-şi putea determina propria distanţă. Această soluţie este, în esenţă, un algoritm serial.

O soluţie paralelă eficientă, necesitând doar un timp $O(\lg n)$, este dată de algoritmul RANG-LISTĂ(L).

În figura 30.2 este ilustrat modul în care algoritmul determină distanţele. Fiecare secţiune a figurii arată starea listei înaintea unei iteraţii a buclei **cât timp** în liniile 6–10. În secţiunea (a) este prezentată lista, imediat după iniţializare. La prima iteraţie, primele 5 obiecte ale listei

nu vor avea pointeri NIL, deci liniile 9–10 vor fi executate de procesoarele corespunzătoare lor. Rezultatul apare în secţiunea (b) a figurii. La a doua iteraţie, numai primele 4 obiecte nu au pointeri NIL: rezultatul acestei iteraţii este prezentat în secţiunea (c). La a treia iteraţie, nu se efectuează decât operaţii asupra primelor două obiecte, şi rezultatul final în care toate obiectele au pointer NIL este prezentat în secţiunea (d).

RANG-LISTĂ(L)
1: **pentru** fiecare procesor i, în paralel **execută**
2: **dacă** $urm[i] = \text{NIL}$ **atunci**
3: $d[i] \leftarrow 0$
4: **altfel**
5: $d[i] \leftarrow 1$
6: **cât timp** există un obiect i astfel încât $urm[i] \neq \text{NIL}$ **execută**
7: **pentru** fiecare procesor i, în paralel **execută**
8: **dacă** $urm[i] \neq \text{NIL}$ **atunci**
9: $d[i] \leftarrow d[i] + d[urm[i]]$
10: $urm[i] \leftarrow urm[urm[i]]$

Ideea implementată în linia 10, în care efectuăm atribuirea $urm[i] \leftarrow urm[urm[i]]$ pentru toţi pointerii $urm[i]$ care nu au valoarea NIL, este numită ***saltul de pointer***. Câmpurile pointer sunt schimbate prin saltul de pointer, deci este distrusă structura listei. Dacă structura listei trebuie păstrată, atunci construim copii ale pointerilor urm; apoi, vom folosi copiile pentru a calcula distanţele.

Corectitudinea

Algoritmului RANG-LISTĂ respectă proprietatea că, la începutul fiecărei iteraţii a buclei **cât timp** din liniile 6–10, pentru fiecare obiect i, dacă adunăm valorile d ale sublistei al cărei vârf este i, obţinem distanţa corectă de la i la capătul listei originale L. De exemplu, în figura 30.2(b), sublista al cărei vârf este obiectul 3 este $\langle 3, 6, 0\rangle$, valorile d corespunzătoare celor trei obiecte din listă, 2, 2 şi 1 având suma 5, adică exact distanţa faţă de sfârşitul listei. Motivul pentru care algoritmul respectă această proprietate este acela că, atunci când fiecare obiect "se lipeşte" de succesorul său din listă, valoarea d a succesorului este adunată la valoarea d a obiectului curent.

Observaţi că, pentru ca algoritmul saltului de pointer să lucreze corect, accesările simultane la memorie trebuie să fie sincronizate. Fiecare execuţie a liniei 10 poate actualiza câţiva pointeri urm. Ne bazăm pe faptul că toate citirile din memorie din partea dreaptă a atribuirii (citirea $urm[urm[i]]$) au loc înaintea oricărei scrieri în memorie (scrierea $urm[i]$) din partea stângă.

Motivul pentru care algoritmul RANG-LISTĂ este un algoritm EREW este acela că fiecare procesor este responsabil pentru cel mult un obiect şi fiecare citire şi scriere din liniile 2–8 este exclusivă, la fel ca şi scrierile din liniile 9 şi 10. Se observă că saltul de pointer respectă proprietatea că, pentru oricare două obiecte distincte i şi j, avem fie $urm[i] \neq urm[j]$, fie $urm[i] = urm[j] = \text{NIL}$. Această proprietate este, cu siguranţă, adevărată pentru lista iniţială şi este păstrată prin execuţia liniei 10. Deoarece toate valorile urm care nu au valoarea NIL sunt distincte, toate citirile din linia 10 sunt exclusive.

Trebuie să presupunem că este efectuată o anume sincronizare în linia 9 pentru ca toate citirile să fie exclusive. Mai exact, este nevoie ca toate procesoarele i să citească, mai întâi, $d[i]$ şi, apoi, $d[urm[i]]$. Dacă un obiect i are valoarea $urm[i] \neq \text{NIL}$ şi există un alt obiect j care

referă i (adică $urm[j] = i$), atunci, cu această sincronizare, prima citire va furniza valoarea $d[i]$ pentru procesorul i, şi a doua va furniza aceeaşi valoare $d[i]$ pentru procesorul j. În concluzie, RANG-LISTĂ este un algoritm EREW.

De aici înainte vom ignora astfel de detalii de sincronizare şi vom presupune că maşinile PRAM şi mediile de programare în pseudocod acţionează într-o manieră sincronizată, toate procesoarele executând citiri şi scrieri în acelaşi timp.

Analiză

Vom arăta acum că, dacă există n obiecte în lista L, atunci RANG-LISTĂ este executată într-un timp $O(\lg n)$. Deoarece iniţializările necesită un timp $O(1)$ şi fiecare iteraţie a buclei **cât timp** necesită un timp $O(1)$, este suficient să arătăm că există exact $\lceil \lg n \rceil$ iteraţii. Observaţia cheie este că fiecare pas al saltului de pointer transformă fiecare listă în două liste "întrepătrunse", una formată din obiectele de rang par şi cealaltă formată din obiectele de rang impar. Deci fiecare pas al saltului de pointer dublează numărul listelor şi înjumătăţeşte lungimile lor. În concluzie, la sfârşitul celor $\lceil \lg n \rceil$ iteraţii, toate listele vor fi formate dintr-un singur obiect.

Presupunem că testul de terminare din linia 6 are nevoie de un timp $O(1)$ datorită reţelei de control din maşina EREW PRAM. Exerciţiul 30.1-8 vă cere să descrieţi o implementare EREW a algoritmului RANG-LISTĂ care realizează explicit testul de terminare. Acest algoritm va trebui să se execute într-un timp $O(\lg n)$.

Pe lângă timpul de execuţie în paralel, există o altă măsură interesantă a performanţei unui algoritm paralel. Definim ***efortul*** depus de un algoritm paralel ca fiind produsul dintre timpul de execuţie şi numărul de procesoare de care are nevoie respectivul algoritm. Intuitiv, efortul este dat de numărul de operaţii pe care le efectuează o maşină RAM serială când simulează algoritmul paralel.

Procedura RANG-LISTĂ depune un efort $\Theta(n \lg n)$ deoarece necesită n procesoare şi un timp de execuţie $\Theta(\lg n)$. Un algoritm serial direct, pentru problema determinării rangului obiectelor unei liste, rulează într-un timp $\Theta(n)$, ceea ce arată că RANG-LISTĂ depune un efort mai mare decât este necesar, dar acest efort este mai mare doar cu un factor logaritmic.

Spunem că un algoritm PRAM A este ***eficient ca efort*** faţă de un alt algoritm (serial sau paralel) B, care rezolvă aceeaşi problemă, dacă efortul depus de A nu este decât cu un factor constant mai mare decât efortul depus de B. Putem spune mai simplu că un algoritm PRAM A este ***eficient ca efort*** dacă este eficient ca efort faţă de cel mai bun algoritm scris pentru o maşină RAM. Deoarece cel mai bun algoritm serial pentru determinarea rangului obiectelor unei liste rulează în timp $\Theta(n)$ pe o maşină serială RAM, RANG-LISTĂ nu este eficient ca efort. În secţiunea 30.4 vom prezenta un algoritm paralel eficient ca efort pentru determinarea rangului obiectelor unei liste.

30.1.2. Determinarea prefixului obiectelor unei liste

Tehnica saltului de pointer poate fi extinsă mult peste determinarea rangului obiectelor unei liste. În secţiunea 29.2.2 se arată modul în care, în contextul circuitelor aritmetice, un calcul al "prefixului" poate fi folosit pentru a realiza, rapid, adunarea binară. Vom cerceta cum poate fi folosit saltul de pointer pentru calculul prefixului. Algoritmul nostru EREW rulează în timp $O(\lg n)$ pe o listă cu n obiecte.

Un ***calcul al prefixului*** este definit în termenii unui operator binar asociativ $\otimes$. Calculul are, ca intrare, un şir $\langle x_1, x_2, \ldots, x_n \rangle$ şi produce, la ieşire, un şir $\langle y_1, y_2, \ldots, y_n \rangle$, astfel încât $y_1 = x_1$ şi $y_k = y_{k-1} \otimes x_k = x_1 \otimes x_2 \otimes \cdots \otimes x_k$ pentru $k = 2, 3, \ldots, n$. Cu alte cuvinte, y_k este obţinut "înmulţind" primele k elemente ale şirului x_k – de aici termenul "prefix". (Definiţia din capitolul 29 indexează şirurile începând cu 0, în timp ce această definiţie le indexează începând cu 1 – o diferenţă neesenţială.)

Pentru exemplificarea calculului prefixului, să presupunem că fiecare element al unei liste cu n obiecte are valoarea 1, şi fie $\otimes$ adunarea obişnuită. Deoarece al k-lea element al listei conţine valoarea $x_k = 1$, pentru $k = 1, 2, \ldots, n$, un calcul al prefixului produce $y_k = k$, indicele celui de-al k-lea element. Deci o altă modalitate de a determina rangul obiectelor unei liste este inversarea listei (care poate fi realizată într-un timp $O(1)$), calculul prefixului şi scăderea unei unităţi din fiecare valoare calculată.

Arătăm, acum, modul în care un algoritm EREW poate calcula, în paralel, prefixele într-un timp $O(\lg n)$ pe o listă cu n obiecte. Fie notaţia

$$[i, j] = x_i \otimes x_{i+1} \otimes \cdots \otimes x_j$$

pentru numerele întregi i şi j din domeniul $1 \le i \le j \le n$. Atunci $[k, k] = x_k$, pentru $k = 1, 2, \ldots, n$, şi

$$[i, k] = [i, j] \otimes [j + 1, k]$$

pentru $1 \le i \le j < k \le n$. Conform acestei notaţii, scopul unui calcul al prefixului este acela de a calcula $y_k = [1, k]$ pentru $k = 1, 2, \ldots, n$.

Când realizăm un calcul de prefix pe o listă, dorim ca ordinea şirului de intrare $\langle x_1, x_2, \ldots, x_n \rangle$ să fie determinată de modul în care sunt înlănţuite obiectele în listă şi nu de indicele obiectelor în şirul din memorie în care sunt stocate obiectele. (Exerciţiul 30.1-2 vă solicită un algoritm care să calculeze prefixul pentru şiruri.) Următorul algoritm EREW începe cu o valoare $x[i]$ pentru fiecare obiect i al listei L. Dacă obiectul i este al k-lea obiect de la începutul listei, atunci $x[i] = x_k$ este al k-lea element al şirului de intrare. Deci calculul paralel al prefixului produce $y[i] = y_k = [1, k]$.

PREFIX-LISTĂ

```
1: pentru fiecare procesor i, în paralel execută
2:    y[i] ← x[i]
3: cât timp există un obiect i astfel încât urm[i] ≠ NIL execută
4:    pentru fiecare procesor i, în paralel execută
5:       dacă urm[i] ≠ NIL atunci
6:          y[urm[i]] ← y[i] ⊗ y[urm[i]]
7:          urm[i] ← urm[urm[i]]
```

Pseudocodul şi figura 30.3 ilustrează similitudinea dintre acest algoritm şi RANG-LISTĂ. Singura diferenţă este iniţializarea şi actualizarea valorilor y. În RANG-LISTĂ, procesorul i actualizează $d[i]$ – propria sa valoare d – în timp ce în PREFIX-LISTĂ, procesorul i actualizează $y[urm[i]]$ – valoarea y a unui alt procesor. Observaţi că PREFIX-LISTĂ este un algoritm EREW din acelaşi motiv ca şi algoritmul RANG-LISTĂ: saltul de pointer respectă proprietatea că pentru obiecte distincte i şi j, fie $urm[i] \ne urm[j]$, fie $urm[i] = urm[j] = \text{NIL}$.

În figura 30.3 este prezentată starea listei înaintea fiecărei iteraţii a buclei **cât timp**. Procedura respectă proprietatea că, la sfârşitul celei de-a t-a execuţie a buclei **cât timp**, al k-lea

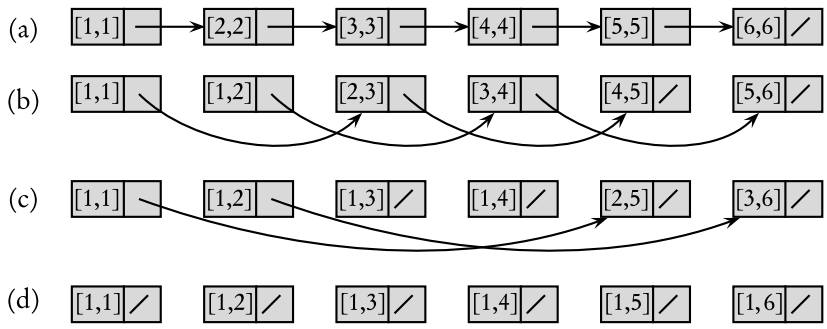


Figura 30.3 Algoritmul paralel pentru determinarea prefixelor obiectelor unei liste, PREFIX-LISTĂ, pe o listă înlănţuită. **(a)** Valoarea iniţială y a celui de-al k-lea obiect din listă este $[k, k]$. Pointerul *urm* al celui de-al k-lea obiect indică spre al $k+1$-lea sau are valoarea NIL pentru ultimul obiect. **(b)-(d)** Valorile y şi *urm* înaintea fiecărui test din linia 3. Răspunsul final apare în partea (d) în care valoarea y pentru al k-lea obiect este $[1, k]$ pentru orice k.

procesor stochează $[\max(1, k - 2^t + 1), k]$, pentru $k = 1, 2, \ldots, n$. La prima iteraţie, al k-lea obiect din listă indică iniţial spre al $(k+1)$-lea obiect, cu excepţia ultimului obiect, care are un pointer NIL. În linia 6, al k-lea obiect, pentru $k = 1, 2, \ldots, n-1$, va extrage valoarea $[k+1, k+1]$ de la succesorul său. Apoi, va efectua operaţia $[k, k] \otimes [k+1, k+1]$ care duce la rezultatul $[k, k+1]$ care va fi stocat în următorul obiect. Se efectuează salturi de pointer pentru pointerii *urm* ca şi în cazul algoritmului RANG-LISTĂ şi rezultatul primei iteraţii apare în figura 30.3(b). În mod similar, putem vedea a doua iteraţie. Pentru $k = 1, 2, \ldots, n-2$, al k-lea obiect extrage valoarea $[k+1, k+2]$ de la succesorul său şi apoi stochează, în următorul obiect, valoarea $[k-1, k] \otimes [k+1, k+2] = [k-1, k+2]$. Rezultatul este arătat în figura 30.3(c). La a treia şi ultima iteraţie, numai primele două obiecte ale listei nu au pointeri cu valoarea NIL şi extrag valori din succesorii lor, afalţi în listele corespunzătoare. Rezultatul final apare în figura 30.3(d). Observaţia cheie care face algoritmul PREFIX-LISTĂ să funcţioneze corect este că, la fiecare pas, dacă efectuăm un calcul al prefixului pe fiecare din listele existente, fiecare obiect obţine valoarea sa corectă.

Deoarece ambii algoritmi folosesc acelaşi mecanism al saltului de pointer, analiza pentru PREFIX-LISTĂ este aceeaşi ca şi RANG-LISTĂ: timpul de execuţie este $O(\lg n)$ pe o maşină EREW PRAM şi efortul total depus este $\Theta(n \lg n)$.

30.1.3. Tehnica ciclului eulerian

În această secţiune, vom introduce tehnica ciclului eulerian şi vom arăta modul în care aceasta poate fi aplicată pentru problema calculării adâncimii fiecărui nod într-un arbore binar cu n noduri. Un pas cheie în acest algoritm EREW care se execută într-un timp $O(\lg n)$ este calculul paralel al prefixului.

Pentru a stoca un arbore binar într-o maşină PRAM, folosim o reprezentare simplă a arborelui binar de genul celei prezentate în secţiunea 11.4. Fiecare nod i are câmpurile $tata[i]$, $stânga[i]$ şi $dreapta[i]$ care indică spre nodul tată al nodului i, respectiv, spre fiul stâng şi spre fiul drept. Să presupunem că fiecare nod este identificat printr-un întreg nenegativ. Din motive care vor fi evidente curând, asociem nu unul, ci trei procesoare pentru fiecare nod şi le vom numi

procesoarele A, B şi C ale nodului respectiv. Ar trebui să putem crea uşor o corespondenţă între un nod şi cele trei procesoare ale sale; de exemplu, nodul i ar putea fi asociat cu procesoarele $3i$, $3i + 1$ şi $3i + 2$.

Calculul adâncimii fiecărui nod al unui arbore cu n noduri necesită un timp $O(n)$ pe o maşină RAM serială. Un algoritm paralel simplu pentru calculul adâncimilor propagă o "undă" în jos de la rădăcina arborelui. Unda ajunge la toate nodurile care au aceeaşi adâncime în acelaşi moment, deci, incrementând un contor, vom putea determina adâncimea fiecărui nod. Acest algoritm este eficient pentru un arbore binar complet deoarece rulează într-un timp proporţional cu înălţimea arborelui. Înălţimea arborelui poate însă avea valoarea $n - 1$, caz în care algoritmul va necesita un timp $\Theta(n)$ – nu este mai bun decât algoritmul serial. Totuşi, folosind tehnica ciclului eulerian, putem calcula adâncimile nodurilor în timp $O(\lg n)$ pe o maşină EREW PRAM, indiferent care este înălţimea arborelui.

Un ***ciclu eulerian*** într-un graf este un ciclu care traversează fiecare muchie exact o dată, cu toate că poate vizita un vârf de mai multe ori. Problema 23-3 vă cerea să demonstraţi că un graf conex orientat conţine un ciclu eulerian dacă şi numai dacă, pentru toate vârfurile v, gradul interior al lui v este egal cu gradul exterior al lui v. Deoarece fiecare muchie (u, v) a unui graf neorientat este pusă în corespondenţă cu două arce (u, v) şi (v, u) din versiunea orientată a grafului, versiunea orientată a oricărui graf conex neorientat – şi deci a oricărui arbore neorientat – are un ciclu eulerian.

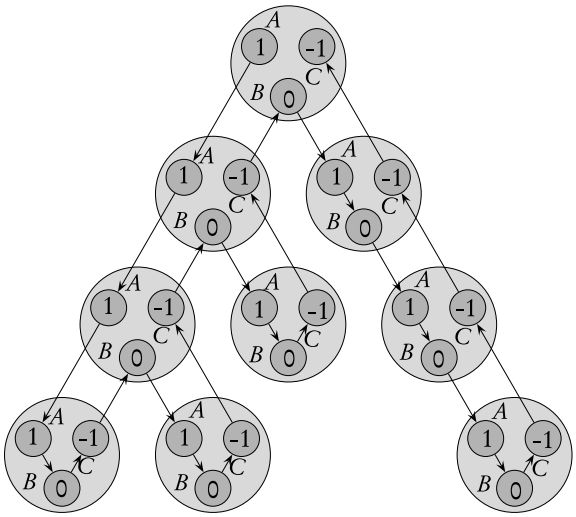
Pentru a calcula adâncimea nodurilor dintr-un arbore binar T, determinăm, mai întâi, un ciclu eulerian al versiunii orientate a lui T (văzut ca un graf neorientat). Ciclul corespunde unei parcurgeri a arborelui şi este reprezentat în figura 30.4(a) printr-o listă înlănţuită care traversează nodurile arborelui. Structura acesteia poate fi descrisă după cum urmează:

- Procesorul A al unui nod indică spre procesorul A al fiului său stâng dacă acesta există sau spre propriul procesor B dacă fiul stâng nu există.
- Procesorul B al unui nod indică spre procesorul A al fiului său drept dacă acesta există sau spre propriul procesor C dacă fiul drept nu există.
- Procesorul C al unui nod indică spre procesorul B al părintelui său dacă este un fiu stâng şi spre procesorul C al părintelui dacă este un fiu drept. Procesorul C al rădăcinii va avea valoarea NIL.

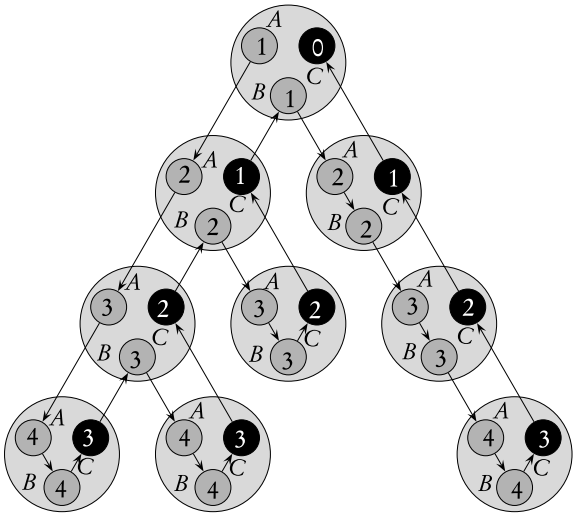
Deci, capul listei înlănţuite formate de ciclul eulerian este procesorul A al rădăcinii şi coada este procesorul C al rădăcinii. Dându-se pointerii care compun arborele original, un ciclu eulerian poate fi construit în timp $O(1)$.

După ce avem lista înlănţuită reprezentând ciclul eulerian al lui T, vom plasa un 1 în fiecare procesor A, un 0 în fiecare procesor B şi un -1 în fiecare procesor C, după cum se arată în figura 30.4(a). Efectuăm, apoi, un calcul paralel al prefixului folosind adunarea obişnuită ca operaţie asociativă aşa cum am făcut şi în secţiunea 30.1.2. În figura 30.4(b) este prezentat rezultatul calculului paralel al prefixului.

După efectuarea calculului paralel al prefixului, adâncimea fiecărui nod va fi păstrată în procesorul C al nodului. De ce? Numerele sunt plasate în procesoarele A, B şi C astfel încât efectul vizitării unui subarbore să fie adunarea unui 0 la valoarea sumei de rulare. Procesorul A al fiecărui nod i contribuie cu 1 la suma de rulare în subarborele stâng al lui i reflectând faptul că adâncimea fiului stâng al lui i este cu 1 mai mare decât adâncimea lui i. Procesorul B contribuie



(a)



(b)

Figura 30.4 Folosirea tehnicii ciclului eulerian pentru calculul adâncimii fiecărui nod dintr-un arbore binar **(a)** Ciclul eulerian este o listă corespunzătoare unei parcurgeri a arborelui. Fiecare procesor conţine un număr folosit de un calcul paralel al prefixului pentru a calcula adâncimile nodurilor. **(b)** Rezultatul calculului paralel al prefixului pe lista înlănţuită de la (a). Procesorul C al fiecărui nod (haşurat cu negru) conţine adâncimea nodului. (Puteţi verifica rezultatul acestui calcul al prefixului calculându-l serial.)

cu 0 deoarece adâncimea fiului stâng al lui i este egală cu adâncimea fiului drept al aceluiaşi nod i. Procesorul C contribuie cu -1 pentru ca, din perspectiva părintelui lui i, întreaga vizită a subarborelui a cărui rădăcină este i să nu aibă efect asupra sumei de rulare.

Lista reprezentând ciclul eulerian poate fi determinată în timp $O(1)$. Lista are $3n$ obiecte, deci calculul paralel al prefixului necesită un timp $O(\lg n)$. În concluzie, timpul total pentru a determina adâncimile tuturor nodurilor este $O(\lg n)$. Deoarece nu sunt necesare accesări concurente ale memoriei, algoritmul prezentat este un algoritm EREW.

Exerciţii

30.1-1 Daţi un algoritm EREW în timp $O(\lg n)$ pentru a decide, pentru fiecare obiect dintr-o listă cu n obiecte, dacă este obiectul din mijloc (al $\lfloor n/2 \rfloor$-lea).

30.1-2 Daţi un algoritm EREW în timp $O(\lg n)$ pentru a efectua calculul prefixului pentru un vector $x[1..n]$. Nu folosiţi pointeri, ci efectuaţi direct calculele de indice.

30.1-3 Să presupunem că fiecare obiect dintr-o listă L cu n obiecte este colorat fie cu roşu, fie cu albastru. Daţi un algoritm EREW eficient pentru a forma două liste care să conţină obiectele din L: una formată din obiectele albastre şi cealaltă din cele roşii.

30.1-4 O maşină EREW PRAM are n obiecte distribuite între câteva liste circulare disjuncte. Daţi un algoritm eficient care determină arbitrar un obiect reprezentativ pentru fiecare listă şi "comunică" fiecărui obiect din listă, identitatea reprezentatului. Presupunem că fiecare procesor îşi cunoaşte propriul indice unic.

30.1-5 Daţi un algoritm EREW care să se execute în timp $O(\lg n)$ pentru a calcula dimensiunile subarborilor care au ca rădăcină cele n noduri ale unui arbore binar. (*Indicaţie:* Luaţi diferenţa a două valori într-o sumă de rulare de-a lungul unui ciclu eulerian.)

30.1-6 Daţi un algoritm EREW eficient pentru determinarea numerotării în preordine, inordine şi postordine a unui arbore binar arbitrar.

30.1-7 Extindeţi tehnica ciclului eulerian de la arbori binari la arbori de sortare cu grade arbitrare ale nodurilor. Mai precis, descrieţi o reprezentare pentru arborii de sortare care permite aplicarea tehnicii ciclului eulerian. Daţi un algoritm EREW pentru a calcula adâncimile nodurilor unui arbore de sortare cu n noduri într-un timp $O(\lg n)$.

30.1-8 Descrieţi o implementare EREW în timp $O(\lg n)$ pentru RANG-LISTĂ care efectuează explicit condiţia de terminare a buclei. (*Indicaţie:* Îmbinaţi testul cu corpul buclei.)

30.2. Algoritmi CRCW şi algoritmi EREW

Discuţiile cu privire la oportunitatea de a oferi posibilitatea accesării concurente a memoriei sunt foarte aprinse. Unii afirmă că mecanismele hardware care suportă algoritmii CRCW sunt prea costisitoare şi sunt folosite prea rar pentru ca dezvoltarea lor să fie justificată. Alţii se plâng că maşinile EREW PRAM furnizează un model de programare prea restrictiv. Probabil

că adevărul este undeva la mijloc şi, de aceea, au fost propuse diferite modele de compromis. Cu toate acestea, este instructiv să examinăm ce avantaje algoritmice sunt oferite de accesările concurente la memorie.

În această secţiune vom arăta că există probleme în care un algoritm CRCW depăşeşte performanţele celui mai bun algoritm EREW posibil. Pentru problema găsirii identităţilor rădăcinilor arborilor dintr-o pădure, citirile concurente permit descrierea unui algoritm mai rapid. Pentru problema găsirii elementului maxim dintr-un vector, scrierile concurente permit descrierea unui algoritm mai rapid.

O problemă în care sunt folositoare citirile concurente

Fie o pădure de arbori binari în care fiecare nod i are un pointer $tata[i]$ care indică spre părintele său şi dorim ca, pentru fiecare nod, să se determine identitatea rădăcinii arborelui din care face parte nodul. Asociind procesorul i cu fiecare nod i din pădurea F, următorul algoritm de salt de pointer stochează identitatea rădăcinii arborelui fiecărui nod i în $rădăcină[i]$.

GĂSEŞTE-RĂDĂCINI

```
1: pentru fiecare procesor i, în paralel execută
2:   dacă tata[i] = NIL atunci
3:     rădăcină[i] ← i
4: cât timp există un nod i astfel încât tata[i] ≠ NIL execută
5:   pentru fiecare procesor i, în paralel execută
6:     dacă tata[i] ≠ NIL atunci
7:       dacă tata[tata[i]] = NIL atunci
8:         rădăcină[i] ← rădăcină[tata[i]]
9:         tata[i] ← tata[tata[i]]
```

În figura 30.5 sunt ilustrate operaţiile efectuate de acest algoritm. După iniţializările efectuate în liniile 1–3, prezentate în figura 30.5(a), singurele noduri care pot identifica rădăcinile corespunzătoare sunt chiar rădăcinile. Bucla **cât timp** din liniile 4–9 execută saltul de pointer şi completează câmpurile *rădăcină*. În figurile 30.5(b)-(d) se prezintă starea pădurii după prima, a doua şi a treia iteraţie a buclei. După cum puteţi vedea, algoritmul respectă proprietatea că, dacă $tata[i] = \text{NIL}$, atunci câmpului $rădăcină[i]$ i se va atribui identitatea nodului rădăcină corespunzător.

Afirmăm că GĂSEŞTE-RĂDĂCINI este un algoritm CRCW care rulează în timp $O(\lg d)$, unde d este adâncimea arborelui cu cea mai mare adâncime din pădure. Singurele scrieri au loc în liniile 3, 8, şi 9 şi acestea sunt exclusive, în fiecare dintre ele procesorul i scriind numai în nodul i. Totuşi, citirile din liniile 8–9 sunt concurente deoarece mai multe noduri pot avea pointeri spre acelaşi nod. De exemplu, în figura 30.5(b) se observă că, în timpul celei de-a doua iteraţii a buclei **cât timp**, $rădăcină[4]$ şi $tata[4]$ sunt citite de procesoarele 18, 2 şi 7.

Timpul de execuţie al algoritmului GĂSEŞTE-RĂDĂCINI este $O(\lg d)$, în principal din acelaşi motiv ca şi RANG-LISTĂ: lungimea fiecărui drum este înjumătăţită la fiecare iteraţie. În figura 30.5 este prezentată, pe larg, această caracteristică.

Cât de rapid pot n noduri dintr-o pădure să determine rădăcinile arborilor binari din care fac parte folosind doar citiri exclusive? O demonstraţie simplă arată că este nevoie de un timp $\Omega(\lg n)$. Observaţia cheie este aceea că, atunci când citirile sunt exclusive, fiecare pas al maşinii PRAM permite unei anumite informaţii să fie copiată în cel mult o altă locaţie de memorie,

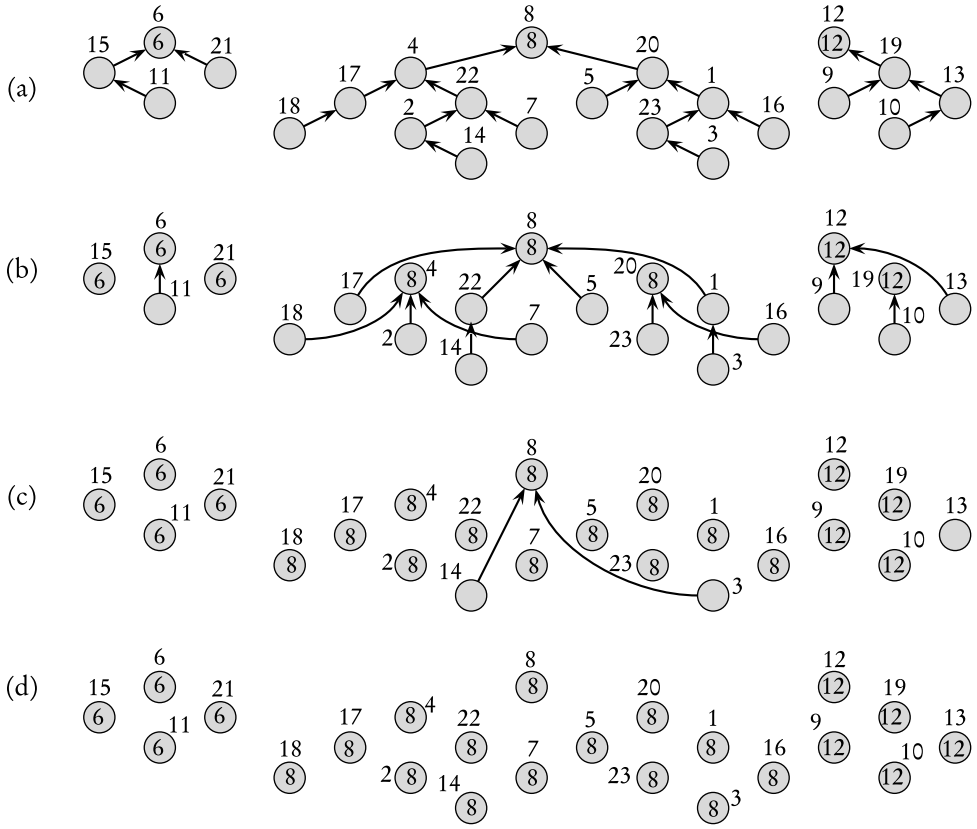


Figura 30.5 Găsirea rădăcinilor într-o pădure de arbori binari pe o maşină CREW PRAM. Indicele nodurilor sunt scrise lângă noduri, iar câmpurile *rădăcină* apar în interiorul nodurilor. Legăturile reprezintă pointerii *tata*. **(a)-(d)** Starea arborilor din pădure de fiecare dată când se execută linia 4 a algoritmului GĂSEŞTE-RĂDĂCINI. Observaţi că lungimile drumurilor sunt înjumătăţite la fiecare iteraţie.

deci numărul locaţiilor care pot conţine o anumită informaţie se poate cel mult dubla la fiecare pas. Luând în considerare un singur arbore, iniţial identitatea rădăcinii este stocată în cel mult o locaţie de memorie. După primul pas cel mult două locaţii pot conţine identitatea rădăcinii, după k paşi cel mult 2^{k-1} locaţii pot conţine identitatea rădăcinii. Dacă dimensiunea arborelui este $\Theta(n)$ avem nevoie ca $\Theta(n)$ locaţii să conţină identitatea rădăcinii când execuţia algoritmului se termină, deci sunt necesari, în total, $\Omega(\lg n)$.

Ori de câte ori adâncimea d a arborelui cu adâncime maximă din pădure este $2^{o(\lg n)}$, algoritmul CREW GĂSEŞTE-RĂDĂCINI depăşeşte asimptotic performanţele oricărui algoritm EREW. Mai exact, pentru orice pădure cu n noduri al cărei arbore de adâncime maximă este un arbore binar echilibrat cu $\Theta(n)$ noduri, $d = O(\lg n)$, caz în care GĂSEŞTE-RĂDĂCINI se execută în timp $O(\lg \lg n)$. Orice algoritm EREW pentru această problemă se execută în timp $\Omega(\lg n)$, adică este, asimptotic, mai lent. Deci citirile concurente ajută în rezolvarea acestei probleme. În exerciţiul 30.2-1 este prezentat un scenariu mai simplu în care citirile concurente sunt folositoare.

O problemă în care sunt folositoare scrierile concurente

Pentru a demonstra că, din punct de vedere al performanţei, scrierile concurente oferă un avantaj faţă de scrierile exclusive, cercetăm problema căutării elementului maxim dintr-un şir de numere reale. Vom vedea că orice algoritm EREW pentru această problemă are nevoie de un timp $\Omega(\lg n)$ şi că nici un algoritm CREW nu are performanţe superioare. Problema poate fi rezolvată în timp $O(1)$, folosind un algoritm CRCW-comun în care, dacă mai multe procesoare scriu în aceeaşi locaţie, toate vor scrie aceeaşi valoare.

Algoritmul CRCW care găseşte maximul dintre cele n elemente ale unui vector presupune că vectorul de intrare este $A[0..n-1]$. Algoritmul foloseşte n^2 procesoare, fiecare procesor comparând $A[i]$ şi $A[j]$ pentru diferite valori i şi j din intervalul $0 \leq i, j \leq n-1$. De fapt, algoritmul efectuează o matrice de comparaţii, deci putem vedea fiecare dintre cele n^2 procesoare ca având nu numai un indice unidimensional în maşina PRAM, ci şi un indice bidimensional (i, j).

```
MAXIM-RAPID
 1: n ← lungime[A]
 2: pentru i ← 0, n − 1, în paralel execută
 3:    m[i] ← ADEVĂRAT
 4: pentru i ← 0, n − 1 şi j ← 0, n − 1, în paralel execută
 5:    dacă A[i] < A[j] atunci
 6:       m[i] ← FALS
 7: pentru i ← 0, n − 1, în paralel execută
 8:    dacă m[i] = ADEVĂRAT atunci
 9:       max ← A[i]
10: returnează max
```

În linia 1 se determină dimensiunea vectorului A; linia trebuie executată numai pe un singur procesor, să zicem procesorul 0. Folosim un vector $m[0..n-1]$, unde procesorul i este responsabil de $m[i]$. Vrem ca $m[i]$ =ADEVĂRAT dacă, şi numai dacă, $A[i]$ este valoarea maximă din vectorul A. Începem (liniile 2–3) presupunând că fiecare element al vectorului poate fi maximul şi ne bazăm pe comparaţiile din linia 5 pentru a determina care elemente ale vectorului nu sunt maxime.

În figura 30.6 este prezentat restul algoritmului. În ciclul din liniile 4–6 verificăm fiecare pereche ordonată de elemente din şirul A. Pentru fiecare pereche $A[i]$ şi $A[j]$ în linia 5, se verifică dacă $A[i] < A[j]$. Dacă rezultatul acestei comparări este ADEVĂRAT, ştim că $A[i]$ nu poate fi maximul, şi în linia 6 se setează $m[i] \leftarrow$ FALS pentru a memora acest fapt. Câteva perechi (i, j) pot scrie în $m[i]$ simultan, dar toate scriu aceeaşi valoare: FALS.

Deci, după execuţia liniei 6, $m[i]$ = ADEVĂRAT pentru exact acei indici i pentru care valoarea $A[i]$ este maximă. Apoi, în liniile 7–9 se stochează valoarea maximă în variabila max, care este returnată în linia 10. Mai multe procesoare pot scrie în locaţia de memorie corespunzătoare variabilei max, dar, dacă acest lucru se întâmplă, toate scriu aceeaşi valoare aşa cum cere modelul CRCW-comun.

Deoarece toate cele trei bucle ale algoritmului sunt executate în paralel, MAXIM-RAPID, se execută în timp $O(1)$. Bineînţeles, algoritmul nu este eficient ca efort deoarece necesită n^2 procesoare şi problema găsirii numărului maxim dintr-un şir poate fi rezolvată într-un timp $\Theta(n)$ de un algoritm serial. Totuşi, putem ajunge mai aproape de un algoritm eficient ca efort, aşa cum vi se cere să arătaţi în exerciţiul 30.2-6

Într-un anumit sens, cheia spre MAXIM-RAPID este că o maşină CRCW PRAM poate să

		$A[j]$					
		5	6	9	2	9	m
$A[i]$	5	F	T	T	F	T	F
	6	F	F	T	F	T	F
	9	F	F	F	F	F	T
	2	T	T	T	F	T	F
	9	F	F	F	F	F	T
							max 9

Figura 30.6 Găsirea maximului a n valori în timp $O(1)$ de către algoritmul CRCW MAXIM-RAPID. Pentru fiecare pereche ordonată de elemente din vectorul de intrare $A = \langle 5, 6, 9, 2, 9 \rangle$, rezultatul comparaţiei $A[i] < A[j]$ este notat în matrice abreviind T pentru ADEVĂRAT şi F pentru FALS. Pentru fiecare linie care conţine o valoare ADEVĂRAT, elementul corespunzător din m este setat pe FALS. Elementele din m care conţin valoarea ADEVĂRAT corespund elementelor cu valoarea maximă din A. În acest caz, valoarea 9 este atribuită variabilei *max*.

efectueze un ŞI logic între n variabile, într-un timp $O(1)$ folosind n procesoare. (Deoarece această posibilitate este valabilă pentru modelul CRCW-comun, ea este valabilă şi pentru modelele CRCW PRAM mai puternice.) Codul efectuează, de fapt, mai multe ŞI-uri în acelaşi timp, calculând pentru $i = 0, 1, \ldots, n-1$,

$$m[i] = \bigwedge_{j=0}^{n-1} (A[i] \geq A[j]),$$

formulă care poate fi dedusă din legile lui DeMorgan (5.2). Această capacitate puternică poate fi folosită şi în alte moduri. De exemplu, capacitatea unei maşini CRCW PRAM de a efectua un ŞI în timp $O(1)$ elimină nevoia ca o reţea separată, de control, să testeze dacă toate procesoarele au terminat iteraţia unui ciclu, aşa cum am presupus pentru algoritmii EREW. Decizia de a termina un ciclu depinde de rezultatul unei operaţii ŞI logic între dorinţele tuturor procesoarelor de a încheia ciclul.

Modelul EREW nu dispune de această puternică facilitate ŞI. Orice algoritm care determină maximul a n elemente are nevoie de un timp $\Omega(\lg n)$. Demonstraţia este, conceptual, similară cu demonstraţia care foloseşte limita inferioară pentru găsirea rădăcinii unui arbore binar. În acea demonstraţie am cercetat câte noduri vor "cunoaşte" identitatea rădăcinii şi am arătat că, la fiecare pas, acest număr este cel mult dublat. Pentru problema determinării maximului a n elemente, considerăm numărul elementelor care "cred" că pot fi maximul. Intuitiv, la fiecare pas al unui algoritm EREW PRAM acest număr se poate, cel mult, înjumătăţi, ceea ce duce la o limită inferioară $\Omega(\lg n)$.

Este remarcabil că limita inferioară $\Omega(\lg n)$ este menţinută chiar dacă permitem citiri concurente, adică este menţinută şi pentru algoritmii CREW. Cook, Dwork, şi Reischuk [50] arată, de fapt, că orice algoritm CREW, pentru găsirea maximului a n elemente, trebuie să se execute în $\Omega(\lg n)$ chiar dacă numărul procesoarelor şi spaţiul de memorie sunt nelimitate. Limita lor inferioară este menţinută şi pentru calculul unui ŞI între n valori logice.

Simularea unui algoritm CRCW cu ajutorul unui algoritm EREW

Ştim, acum, că algoritmii CRCW pot rezolva unele probleme mai rapid decât algoritmii EREW. Mai mult, orice algoritm EREW poate fi executat pe o maşină CRCW PRAM. Deci modelul CRCW este mai puternic decât modelul EREW. Dar cu cât este mai puternic? În secţiunea 30.3, vom arăta că o maşină EREW PRAM cu p procesoare poate sorta p numere în timp $O(\lg p)$. Folosim acum acest rezultat pentru a furniza o limită superioară teoretică a puterii unei maşini CRCW PRAM faţă de o maşină EREW PRAM.

Teorema 30.1 Un algoritm CRCW care are la dispoziţie p procesoare nu poate fi mai rapid decât de, cel mult, $O(\lg p)$ ori decât cel mai bun algoritm EREW care are la dispoziţie tot p procesoare şi rezolvă aceeaşi problemă.

Demonstraţie. Demonstraţia se face prin simulare. Simulăm fiecare pas al algoritmului CRCW printr-un calcul EREW în timp $O(\lg p)$. Deoarece puterea de procesare a ambelor maşini este aceeaşi, nu trebuie să ne concentrăm decât pe accesările de memorie. Vom prezenta numai demonstraţia pentru simularea scrierilor concurente. Implementarea citirilor concurente este lăsată pe seama cititorului (exerciţiul 30.2-8).

Cele p procesoare ale maşinii EREW PRAM simulează o scriere concurentă a algoritmului CRCW folosind un vector auxiliar A de dimensiune p. Această idee este ilustrată în figura 30.7. Când procesoarele CRCW P_i, pentru $i = 0, 1, \ldots, p-1$, doresc să scrie o dată x_i în locaţia l_i, fiecare procesor EREW P_i corespunzător scrie perechea ordonată (l_i, x_i) în locaţia $A[i]$. Aceste scrieri sunt exclusive deoarece fiecare procesor scrie într-o locaţie de memorie distinctă. Apoi, şirul A este ordonat după prima coordonată a perechii ordonate în timp $O(\lg p)$, ceea ce determină ca toate datele care trebuie scrise în aceeaşi locaţie să fie aduse una lângă cealaltă la ieşire.

Fiecare procesor EREW P_i, pentru $i = 1, 2, \ldots, p-1$, cercetează, acum, $A[i] = (l_j, x_j)$ şi $A[i-1] = (l_k, x_k)$, unde j şi k sunt valori din domeniul $0 \le j, k \le p-1$. Dacă $l_j \ne l_k$ sau $i = 0$, atunci procesorul P_i pentru $i = 0, 1, \ldots, p-1$, scrie data x_j în locaţia l_j din memoria globală. În caz contrar, procesorul nu execută nimic. Deoarece vectorul A este ordonat după prima coordonată, numai unul dintre procesoare reuşeşte să scrie la o locaţie dată, deci scrierile sunt exclusive. În concluzie, acest proces implementează fiecare pas al scrierilor concurente în modelul CRCW-comun în timp $O(\lg p)$. ∎

Alte modele pentru scrierea concurentă pot fi, de asemenea, simulate. (Vezi exerciţiul 30.2-9.)

Ca urmare a acestui fapt, apare întrebarea ce model este preferabil – CRCW sau EREW – şi dacă răspunsul este CRCW, care dintre modelele CRCW. Adepţii modelului CRCW afirmă că acesta este mai uşor de programat decât modelul EREW şi că algoritmii CRCW sunt mai rapizi. Criticii afirmă că hardware-ul care implementează operaţiile concurente cu memoria este mai lent decât hardware-ul care implementează operaţiile exclusive cu memoria, şi deci timpul de execuţie mai rapid al algoritmilor CRCW este fictiv. În realitate, spun criticii, nu putem determina maximul a n valori în timp $O(1)$.

Alţii spun că maşinile PRAM – fie CRCW, fie EREW – nu reprezintă un model potrivit, că procesoarele trebuie interconectate printr-o reţea de comunicaţie, şi că reţeaua de comunicaţie ar trebui să facă parte din model. Procesoarele ar trebui să poată comunica numai cu vecinii lor din reţea.

Este destul de clar că problema modelului paralel potrivit nu îşi va găsi, prea uşor, o rezolvare în favoarea vreunui model. Totuşi, un fapt important, de care trebuie să ne dăm seama, este că aceste modele sunt numai atât: modele. Pentru o situaţie reală, diferite modele pot fi aplicate în

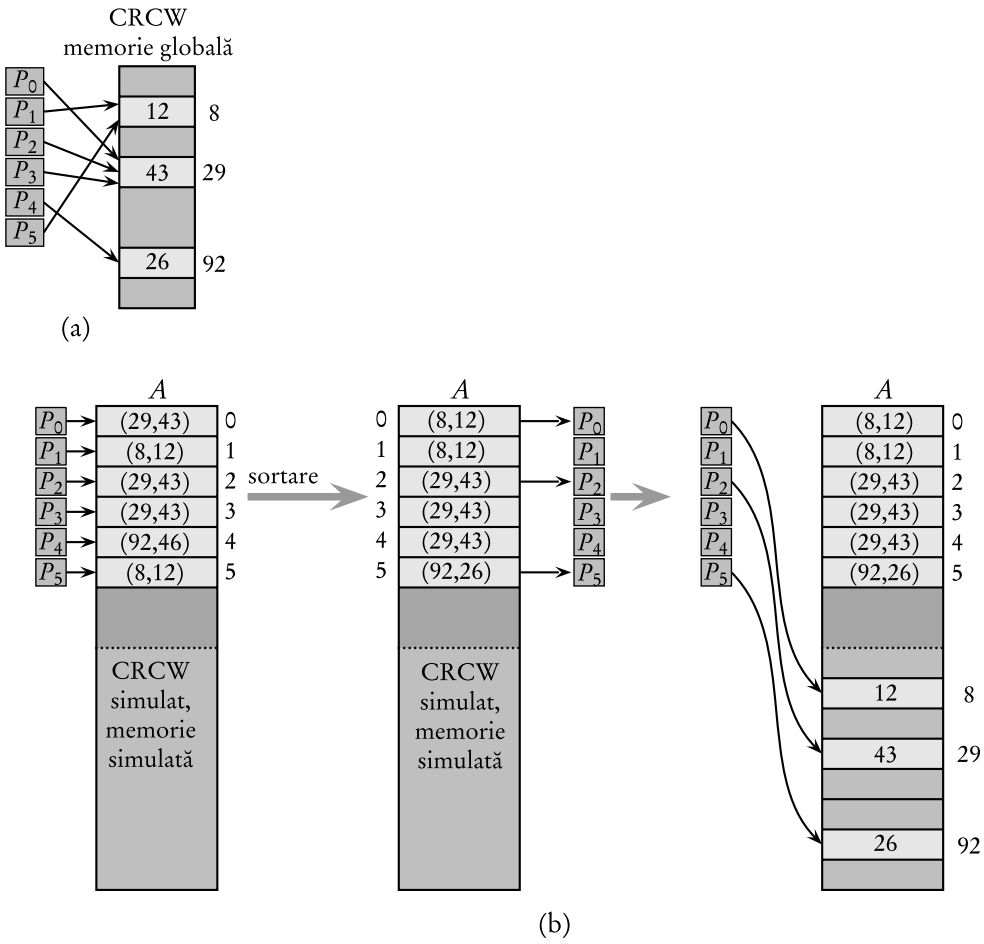


Figura 30.7 Simularea scrierilor concurente pe o maşină EREW PRAM. **(a)** Un pas al algoritmului CRCW-comun în care 6 procesoare efectuează scrieri concurente în memoria globală. **(b)** Simularea pasului pe o maşină EREW PRAM. La început, perechile ordonate care conţin locaţiile şi datele sunt scrise într-un vector A care este apoi sortat. Comparând elementele adiacente ale vectorului, ne asigurăm că numai prima dintre un grup de scrieri identice în memoria globală, este implementată. În acest caz procesoarele P_0, P_2 şi P_5 realizează scrierile.

anumite proporţii. Gradul în care un anumit model se potriveşte cu situaţia tehnică este gradul în care analizele algoritmice, folosind respectivul model, vor prezice fenomenele din lumea reală. Este important să studiem diferite modele paralele şi diferiţi algoritmi paraleli pentru că, pe măsură ce domeniul calculului paralel se va extinde, probabil se va lămuri care paradigme ale calculului paralel sunt mai potrivite pentru implementare.

Exerciţii

30.2-1 Să presupunem că ştim că o pădure de arbori binari constă dintr-un singur arbore cu n noduri. Arătaţi că, luând în considerare această ipoteză, o implementare CREW a algoritmului GĂSEŞTE-RĂDĂCINI poate rula în timp $O(1)$, independent de adâncimea arborelui. Demonstraţi că orice algoritm EREW are nevoie de un timp $\Omega(\lg n)$.

30.2-2 Daţi un algoritm EREW pentru GĂSEŞTE-RĂDĂCINI care se execută în timp $O(\lg n)$ pe o pădure cu n noduri.

30.2-3 Daţi un algoritm CRCW care are la dispoziţie n procesoare şi care calculează un SAU între n valori logice în timp $O(1)$.

30.2-4 Descrieţi un algoritm CRCW eficient pentru a înmulţi două matrice logice de dimensiuni $n \times n$ folosind n^3 procesoare.

30.2-5 Descrieţi un algoritm EREW în timp $O(\lg n)$ pentru a înmulţi două matrice de numere reale folosind n^3 procesoare. Există un algoritm CRCW-comun mai rapid? Există un algoritm mai rapid într-unul din modelele CRCW mai puternice?

30.2-6 ⋆ Demonstraţi că, pentru orice constantă $\epsilon > 0$, există un algoritm CRCW în timp O(1) care, folosind $O(n^{1+\epsilon})$ procesoare, determină elementul maxim al unui vector cu n elemente.

30.2-7 ⋆ Arătaţi cum pot fi interclasaţi doi vectori, fiecare format din n elemente, în timp $O(1)$ folosind un algoritm CRCW-prioritar. Descrieţi modul în care acest algoritm se poate folosi pentru a sorta în timp $O(\lg n)$. Este algoritmul pe care l-aţi propus eficient ca efort?

30.2-8 Completaţi demonstraţia teoremei 30.1 descriind modul în care o citire concurentă pe o maşină CRCW PRAM având p procesoare este implementată în timp $O(\lg p)$ pe o maşină EREW PRAM având p procesoare.

30.2-9 Arătaţi cum o maşină EREW PRAM cu p procesoare poate implementa un algoritm pentru CRCW-combinat PRAM cu o pierdere de performanţă de numai $O(\lg p)$. (*Indicaţie:* Folosiţi un calcul paralel al prefixului.)

30.3. Teorema lui Brent şi eficienţa efortului

Teorema lui Brent arată cum putem simula eficient un circuit combinaţional printr-un PRAM. Folosind această teoremă, putem adapta modelului PRAM multe rezultate pentru reţele de sortare din capitolul 28 şi foarte multe rezultate pentru circuite aritmetice din capitolul 29. Cititorii nefamiliarizaţi cu circuitele combinaţionale pot să revadă secţiunea 29.1.

Un ***circuit combinaţional*** este o reţea aciclică de ***elemente combinaţionale***. Fiecare element combinaţional are una sau mai multe intrări şi, în această secţiune, vom presupune că fiecare element are exact o ieşire. (Elementele combinaţionale cu $k > 1$ ieşiri pot fi considerate ca fiind k elemente separate). Numărul intrărilor este ***evantaiul de intrare*** al elementului, iar numărul ieşirilor sale este ***evantaiul de ieşire***. În general, în această secţiune, presupunem că

orice element combinaţional din circuit are o intrare mărginită ($O(1)$). Un element poate avea, totuşi, o ieşire nemărginită.

Dimensiunea unui circuit combinaţional este numărul elementelor combinaţionale pe care le conţine. Numărul elementelor combinaţionale de pe un cel mai lung drum de la o intrare a circuitului până la o ieşire a unui element combinaţional este ***adâncimea*** elementului. ***Adâncimea*** întregului circuit este adâncimea maximă a oricărui element al său.

Teorema 30.2 (Teorema lui Brent) Orice circuit combinaţional de adâncime d şi dimensiune n cu intrarea mărginită poate fi simulat printr-un algoritm CREW cu p procesoare într-un timp $O(n/p + d)$.

Demonstraţie. Memorăm intrările circuitului combinaţional în memoria globală a PRAM-ului şi rezervăm o locaţie de memorie pentru fiecare element combinaţional din circuit pentru a păstra valoarea sa de ieşire, atunci când aceasta este calculată. Astfel, un element combinaţional dat poate fi simulat printr-un singur procesor PRAM într-un timp $O(1)$, după cum se arată în continuare. Procesorul citeşte, pur şi simplu, valorile de intrare pentru un element din valorile din memorie, care corespund intrărilor circuitului sau ieşirilor elementului care îl alimentează, simulând astfel firele din circuit. Acesta calculează apoi funcţia elementului combinaţional şi scrie rezultatul în memorie, la poziţia corespunzătoare. Deoarece numărul de intrări ale fiecărui element din circuit este mărginit, fiecare funcţie poate fi calculată într-un timp $O(1)$.

De aceea, sarcina noastră este de a găsi o planificare pentru cele p procesoare ale PRAM-ului, astfel încât toate elementele combinaţionale să fie simulate într-un timp $O(n/p + d)$. Principala dificultate este că un element nu poate fi simulat până când nu sunt calculate ieşirile oricărui element care îl alimentează. Sunt folosite citiri concurente ori de câte ori mai multe elemente combinaţionale ce sunt simulate în paralel au nevoie de aceeaşi valoare.

Deoarece toate elementele de la adâncimea 1 depind doar de intrările circuitului, ele sunt singurele care pot fi simulate iniţial. După ce acestea au fost simulate, putem simula toate elementele de la adâncimea 2 şi aşa mai departe, până când terminăm cu toate elementele de la adâncimea d. Ideea principală este că, dacă au fost simulate toate elementele de la adâncimea 1 până la adâncimea i, putem simula orice submulţime de elemente de la adâncimea $i + 1$, în paralel, din moment ce calculele necesare nu depind unul de altul.

De aceea, strategia de planificare este destul de naivă. Simulăm toate elementele de la adâncimea i, înainte de a începe să le simulăm pe cele de la adâncimea $i + 1$. În cadrul unei adâncimi i date, simulăm p elemente o dată. Figura 30.8 ilustrează o astfel de strategie pentru $p = 2$.

Vom analiza acum această strategie de simulare. Pentru $i = 1, 2, \ldots, d$, fie n_i numărul elementelor de la adâncimea i din circuit. Astfel,

$$\sum_{i=1}^{d} n_i = n.$$

Să considerăm cele n_i elemente combinaţionale de la adâncimea i. Grupându-le în $\lceil n_i/p \rceil$ grupuri, unde primele $\lfloor n_i/p \rfloor$ grupuri au câte p elemente fiecare, iar elementele rămase, dacă există, se află în ultimul grup, PRAM-ul poate simula calculele efectuate de către aceste elemente combinaţionale într-un timp $O(\lceil n_i/p \rceil)$. Timpul total pentru simulare este atunci de ordinul

$$\sum_{i=1}^{d} \left\lceil \frac{n_i}{p} \right\rceil \leq \sum_{i=1}^{d} \left(\frac{n_i}{p} + 1 \right) = \frac{n}{p} + d.$$

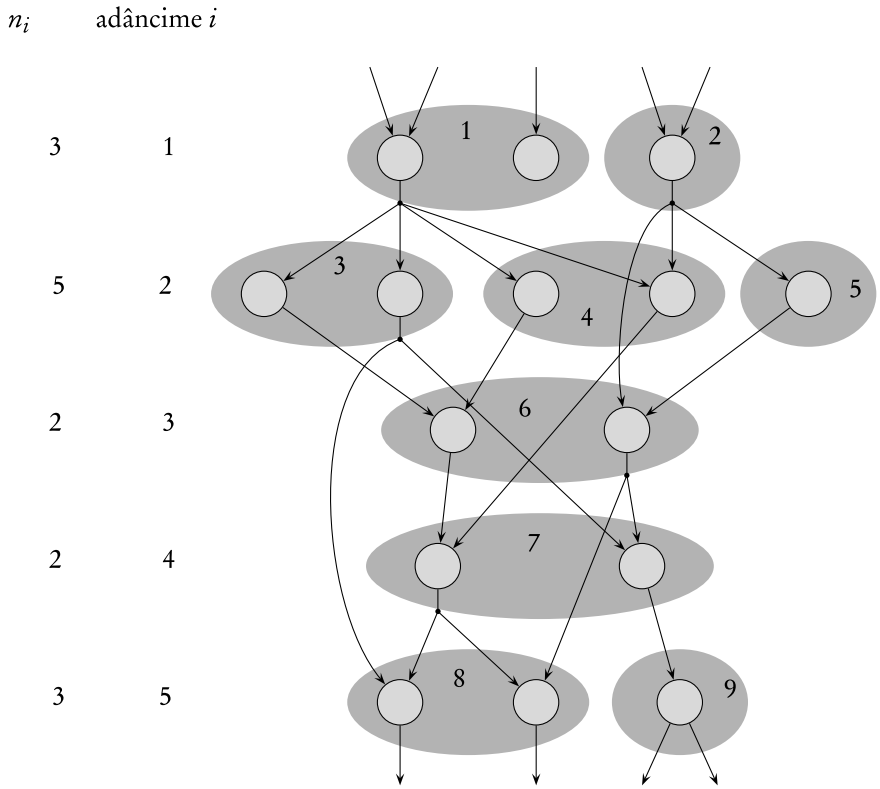


Figura 30.8 Teorema lui Brent. Circuitul combinaţional de dimensiune 15 şi adâncime 5 este simulat printr-un PRAM CREW în $9 \leq 15/2 + 5$ paşi. Simularea se efectuează de sus în jos, de-a lungul circuitului. Grupurile haşurate de elemente ale circuitului indică elementele simulate în acelaşi moment, şi fiecare grup este etichetat cu un număr corespunzator momentului când elementele sale sunt simulate.

■

Teorema lui Brent poate fi extinsă la simulări EREW atunci când un circuit combinaţional are un număr de ieşiri de ordinul $O(1)$ pentru fiecare element combinaţional.

Corolarul 30.3 Orice circuit combinaţional de adâncime d şi dimensiune n, având numărul de intrări şi ieşiri mărginit, poate fi simulat pe un PRAM EREW cu p procesoare într-un timp $O(n/p + d)$.

Demonstraţie. Vom efectua o simulare similară celei din demonstraţia teoremei lui Brent. Singura diferenţă constă în simularea firelor, unde teorema 30.2 necesită citiri concurente. Pentru simularea EREW, după ce ieşirea unui element combinaţional este calculată, ea nu este citită direct de către procesoarele, care au nevoie de valoarea sa. În loc de aceasta, valoarea ieşirii este copiată de către procesorul ce simulează elementul în intrările de dimensiune $O(1)$ care au nevoie de ea. Procesoarele care au nevoie de valoare, o pot citi fără a crea interferenţe între ele. ■

Această strategie de simulare EREW nu funcţionează pentru elemente cu număr de ieşiri

nemărginit, deoarece, la fiecare pas, copierea poate consuma un timp mai mare decât cel constant. Astfel, pentru circuite care au elemente cu număr de ieşiri nemărginit, avem nevoie de puterea citirilor concurente. (Cazul numărului de intrări nemărginit poate fi, uneori, tratat printr-o simulare CRCW dacă elementele combinaţionale sunt destul de simple. Vedeţi exerciţiul 30.3-1.)

Corolarul 30.3 ne oferă un algoritm de sortare EREW rapid. După cum s-a explicat în notele bibliografice ale capitolului 28, reţeaua de sortare AKS poate sorta n numere într-o adâncime $O(\lg n)$ folosind $O(n \lg n)$ comparatori. Deoarece comparatorii au numărul de intrări limitat, există un algoritm EREW pentru sortarea a n numere într-un timp $O(\lg n)$ folosind n procesoare. (Am folosit acest rezultat în teorema 30.1 pentru a demonstra că un PRAM EREW poate simula un PRAM CRCW cu o încetinire logaritmică, în cel mai defavorabil caz.) Din nefericire, constantele ascunse de către notaţia O sunt atât de mari, încât acest algoritm de sortare nu prezintă decât un interes teoretic. Totuşi, au fost descoperiţi algoritmi de sortare EREW mai practici, în mod notabil algoritmul de sortare prin interclasare paralel, datorat lui Cole [46].

Acum, să presupunem că avem un algoritm PRAM care foloseşte cel mult p procesoare, dar avem un PRAM cu numai $p' < p$ procesoare. Am dori să putem executa algoritmul pentru p procesoare pe PRAM-ul mai mic cu numai p' procesoare într-o manieră eficientă. Folosind ideea din demonstraţia teoremei lui Brent, putem da o condiţie pentru a stabili când este posibil acest lucru.

Teorema 30.4 Dacă un algoritm PRAM A pentru p procesoare se execută într-un timp t, atunci, pentru orice $p' < p$, există un algoritm PRAM A' pentru p' procesoare care se execută într-un timp $O(pt/p')$.

Demonstraţie. Să presupunem că paşii algoritmului A sunt numerotaţi cu $1, 2, \ldots, t$. Algoritmul A' simulează execuţia fiecărui pas $i = 1, 2, \ldots, t$ într-un timp $O(\lceil p/p' \rceil)$. Există t paşi şi, astfel, întreaga simulare consumă un timp $O(\lceil p/p' \rceil t) = O(pt/p')$ din moment ce $p' < p$. ∎

Efortul efectuat de algoritmul A este pt, iar efortul efectuat de algoritmul A' este $(pt/p')p' = pt$. De aceea, simularea este eficientă ca efort. Prin urmare, dacă algoritmul A este el însuşi eficient ca efort, atunci şi algoritmul A' este eficient.

De aceea, atunci când dezvoltăm algoritmi eficienţi ca efort pentru o problemă, nu trebuie să creăm un algoritm nou pentru fiecare număr diferit de procesoare. De exemplu, să presupunem că putem demonstra o limită inferioară strânsă t a timpului de execuţie al oricărui algoritm paralel, indiferent pentru câte procesoare, pentru a rezolva o problemă dată, şi să presupunem, în continuare, că cel mai bun algoritm serial pentru acea problemă efectuează efortul w. Atunci, nu trebuie decât să dezvoltăm un algoritm eficient ca efort pentru problema care foloseşte $p = \Theta(w/t)$ procesoare pentru a obţine algoritmi eficienţi ca efort pentru orice număr de procesoare pentru care un algoritm eficient ca efort este posibil. Pentru $p' = o(p)$, teorema 30.4 garantează că există un algoritm eficient. Pentru $p' = \omega(p)$, nu există nici un algoritm eficient deoarece, dacă t este o limită inferioară a timpului pentru orice algoritm paralel, $p't = \omega(pt) = \omega(w)$.

Exerciţii

30.3-1 Demonstraţi un rezultat analog teoremei lui Brent pentru o simulare CRCW a unor circuite combinaţionale booleene, având porţi ŞI şi SAU cu un număr nelimitat de intrări. (*Indicaţie:* Consideraţi că "dimensiunea" este numărul total de intrări la porţile din circuit.)

30.3-2 Demonstraţi că un calcul paralel al prefixelor, efectuat pe n valori memorate într-un tablou de memorie, poate fi implementat într-un timp $O(\lg n)$ pe un PRAM EREW folosind $O(n/\lg n)$ procesoare. De ce acest rezultat nu poate fi extins automat la o listă de n valori?

30.3-3 Arătaţi cum se poate înmulţi o matrice A de dimensiuni $n \times n$ cu un vector b de dimensiune n într-un timp $O(\lg n)$ cu ajutorul unui algoritm EREW eficient. (*Indicaţie:* Construiţi un circuit combinaţional pentru problemă.)

30.3-4 Daţi un algoritm CRCW care foloseşte n^2 procesoare pentru a înmulţi două matrice de dimensiuni $n \times n$. Algoritmul trebuie să fie eficient ca efort în raport cu timpul obişnuit $\Theta(n^3)$ necesar algoritmului serial de înmulţire a matricelor. Puteţi realiza algoritmul EREW?

30.3-5 Unele modele paralele permit procesoarelor să devină inactive, astfel încât numărul procesoarelor care funcţionează variază la fiecare pas. Se defineşte efortul, în acest model, ca fiind numărul total de paşi executaţi în timpul unui algoritm de către procesoarele active. Demonstraţi că orice algoritm CRCW care efectuează un efort w şi rulează într-un timp t poate fi rulat pe un PRAM EREW într-un timp $O((w/p + t)\lg p)$. (*Indicaţie:* Partea dificilă este planificarea procesoarelor active *în timp* ce se efectuează calculul.)

30.4. Calculul paralel de prefix, eficient ca efort

În secţiunea 30.1.2, am examinat un algoritm EREW LISTEAZĂ-RANG de timp $O(\lg n)$ care poate executa un calcul de prefix asupra unei liste înlănţuite de n obiecte. Algoritmul foloseşte n procesoare şi efectuează un efort $\Theta(n \lg n)$. Deoarece putem executa uşor un calcul de prefix într-un timp $\Theta(n)$ pe un calculator serial, LISTEAZĂ-RANG nu este eficient.

Această secţiune prezintă un algoritm EREW paralel, aleator, de calculare a prefixului care este eficient ca efort. Algoritmul foloseşte $\Theta(n/\lg n)$ procesoare, şi se execută, cu o mare probabilitate, într-un timp $O(\lg n)$. Astfel, foarte probabil, el este eficient ca efort. Mai mult, cu ajutorul teoremei 30.4, putem construi imediat, pe baza acestui algoritm, algoritmi eficienţi pentru orice număr de procesoare $p = O(n/\lg n)$.

Calcul paralel, recursiv, de prefix

Algoritmul paralel, recursiv, pentru calcularea prefixului LISTARE-ALEATOARE-DE-PREFIX lucrează pe o listă înlănţuită de n obiecte, folosind $p = \Theta(n/\lg n)$ procesoare. În timpul execuţiei algoritmului, fiecare procesor este responsabil pentru $n/p = \Theta(\lg n)$ din obiectele din lista originală. Obiectele sunt repartizate procesoarelor în mod arbitrar (nu neapărat în secvenţe contigue) înainte ca apelurile recursive să aibă loc, iar "posesorii" obiectelor nu se schimbă niciodată. Pentru uşurinţă, presupunem că lista este dublu înlănţuită deoarece crearea legăturilor duble pentru o singură listă necesită un timp $O(1)$.

Ideea algoritmului LISTARE-ALEATOARE-DE-PREFIX este să eliminăm o parte din obiectele din listă, să efectuăm, apoi, un calcul recursiv de prefix asupra listei rezultate, şi pe urmă să o dezvoltăm, reunind-o cu obiectele eliminate, pentru a obţine un calcul de prefix al listei originale. Figura 30.9 ilustrează procesul recursiv, iar figura 30.10 ilustrează cum se desfăşoară apelurile recursive. Vom demonstra puţin mai târziu că fiecare pas al recursivităţii satisface următoarele proprietăţi:

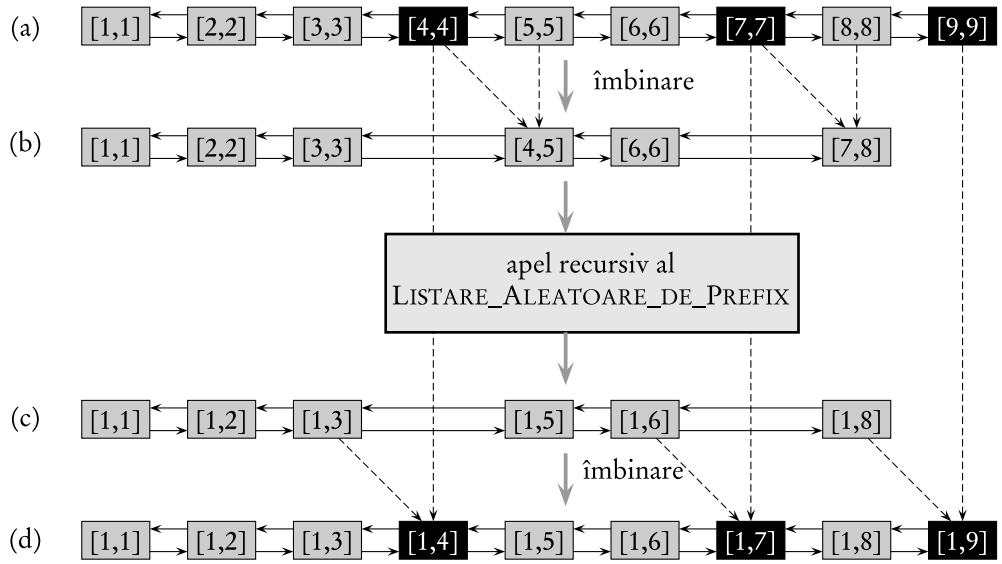


Figura 30.9 Algoritmul paralel, recursiv, aleator şi eficient ca efort LISTARE-ALEATOARE-DE-PREFIX pentru efectuarea unui calcul de prefix pentru o listă înlănţuită de $n = 9$ obiecte. **(a)-(b)** O mulţime de obiecte neadiacente (desenate cu negru) sunt selectate pentru a fi eliminate. Valoarea din fiecare obiect negru este folosită pentru recalcularea valorii din următorul obiect din listă, iar apoi obiectul negru este eliminat, fiind unit cu obiectul următor. Algoritmul este apelat recursiv pentru calcularea unui prefix paralel pentru lista contractată. **(c)-(d)** Valorile rezultate reprezintă valorile finale corecte pentru obiectele din lista contractată. Obiectele eliminate sunt, apoi, reintroduse în listă şi fiecare foloseşte valoarea obiectului precedent pentru a-şi calcula valoarea sa finală.

1. Cel mult un obiect, dintre cele care aparţin unui procesor dat, este selectat pentru eliminare.

2. Niciodată, nu sunt selectate, pentru eliminare, două obiecte adiacente.

Înainte de a arăta cum putem selecta obiecte care satisfac aceste proprietăţi, să examinăm mai detaliat modul în care se efectuează calculul de prefix. Să presupunem că, la fiecare pas al recursivităţii, al k-lea obiect din listă este selectat pentru eliminare. Acest obiect conţine valoarea $[k, k]$, care este adusă de cel de-al $(k + 1)$-lea obiect din listă. (Situaţiile limită, de exemplu cea de aici, când k se află la sfârşitul listei, pot fi tratate direct şi nu sunt descrise.) Al $(k + 1)$-lea obiect, care memorează valoarea $[k + 1, k + 1]$, este apoi calculat şi va conţine $[k, k + 1] = [k, k] \otimes [k + 1, k + 1]$. Al k-lea obiect este eliminat, apoi, din listă, fiind unit cu obiectul următor.

Apoi, procedura LISTARE-ALEATOARE-DE-PREFIX se autoapelează pentru a executa un calcul de prefix pentru lista "contractată". (Apelurile recursive se termină atunci când lista este vidă.) Observaţia cheie este că, atunci când se revine din apelul recursiv, fiecare obiect din lista contractată conţine valoarea corectă de care are nevoie pentru calculul paralel de prefix pentru lista iniţială. Tot ce rămâne de făcut este să reinserăm în listă obiectele eliminate anterior, cum este, de exemplu, al k-lea obiect, şi să recalculam valorile lor.

După ce al k-lea obiect este reintrodus în listă, valoarea finală a prefixului său poate fi

calculată folosind valoarea din cel de-al $(k-1)$-lea obiect. După apelul recursiv, al $(k-1)$-lea obiect conţine $[1, k-1]$ şi, astfel, al k-lea obiect – care conţine încă valoarea $[k, k]$ – nu are nevoie decât de valoarea $[1, k-1]$ şi să calculeze $[1, k] = [1, k-1] \otimes [k, k]$.

Datorită proprietăţii 1, fiecare obiect selectat dispune de un procesor distinct pentru a efectua operaţiile necesare pentru eliminarea sau reintroducea obiectului în listă. Proprietatea 2 asigură faptul că nu apare nici o confuzie între procesoare, atunci când se elimină şi se reintroduc obiecte (vezi exerciţiul 30.4-1). Împreună, cele două proprietăţi asigură faptul că fiecare pas al recursivităţii poate fi implementat într-un mod EREW într-un timp $O(1)$.

Selectarea obiectelor pentru eliminare

Cum selectează procedura LISTARE-ALEATOARE-DE-PREFIX obiectele pe care le elimină? Ea trebuie să satisfacă cele două proprietăţi enunţate mai sus şi, în lpus, dorim ca timpul necesar selectării obiectelor să fie scurt (şi, preferabil, constant). Mai mult, dorim să fie selectate cât mai multe obiecte posibil.

Următoarea metodă de selectare aleatoare satisface aceste condiţii. Obiectele sunt selectate prin programarea fiecărui procesor astfel încât el să execute următorii paşi:

1. Procesorul alege un obiect i, care nu a fost selectat anterior, dintre cele pe care le deţine.
2. Apoi, procesorul "dă cu banul", alegând valorile CAP sau BAN cu aceeaşi probabilitate.
3. Dacă a ales CAP, procesorul marchează obiectul i ca SELECTAT, dacă $urm[i]$ nu a fost ales de către un alt procesor a cărui monedă este, de asemenea, CAP.

Această metodă aleatoare necesită doar un timp $O(1)$ pentru a selecta obiecte pentru eliminare, şi nu necesită accesări concurente de memorie.

Trebuie să demonstrăm că această procedură satisface cele două proprietăţi de mai sus. Faptul că proprietatea 1 este adevarată se vede foarte uşor deoarece doar un singur obiect este ales de către un procesor pentru o posibilă selectare. Pentru a arăta că proprietatea 2 este adevărată, să presupunem contrariul, şi anume că două obiecte consecutive, i şi $urm[i]$, sunt selectate. Această situaţie apare doar dacă ambele obiecte au fost alese de către procesoarele lor şi ambele procesoare au obţinut CAP la aruncarea monezii. Dar obiectul i nu este selectat dacă procesorul responsabil pentru $urm[i]$ a obţinut CAP, ceea ce este o contradicţie.

Analiză

Deoarece fiecare pas recursiv al algoritmului LISTARE-ALEATOARE-DE-PREFIX se execută într-un timp $O(1)$, pentru a analiza algoritmul, nu trebuie decât să determinăm de câţi paşi are acesta nevoie pentru a elimina toate obiectele din lista iniţială. La fiecare pas, un procesor are o probabilitate de cel puţin $1/4$ de a elimina obiectul i pe care îl alege. De ce? Pentru că, atunci când se aruncă moneda, rezultă CAP cu o probabilitate de cel puţin $1/2$, iar probabilitatea ca fie să nu aleagă $urm[1]$, fie să îl aleagă şi să obţină BAN, atunci când se aruncă moneda, este de cel puţin $1/2$. Din moment ce aceste două aruncări de monedă sunt două evenimente independente, putem înmulţi probabilităţile lor, obţinând o probabilitate de cel puţin $1/4$ ca un procesor să elimine obiectul pe care îl alege. Deoarece fiecare procesor deţine $\Theta(\lg n)$ obiecte, timpul aşteptat în care un procesor îşi elimină toate obiectele sale este $\Theta(\lg n)$.

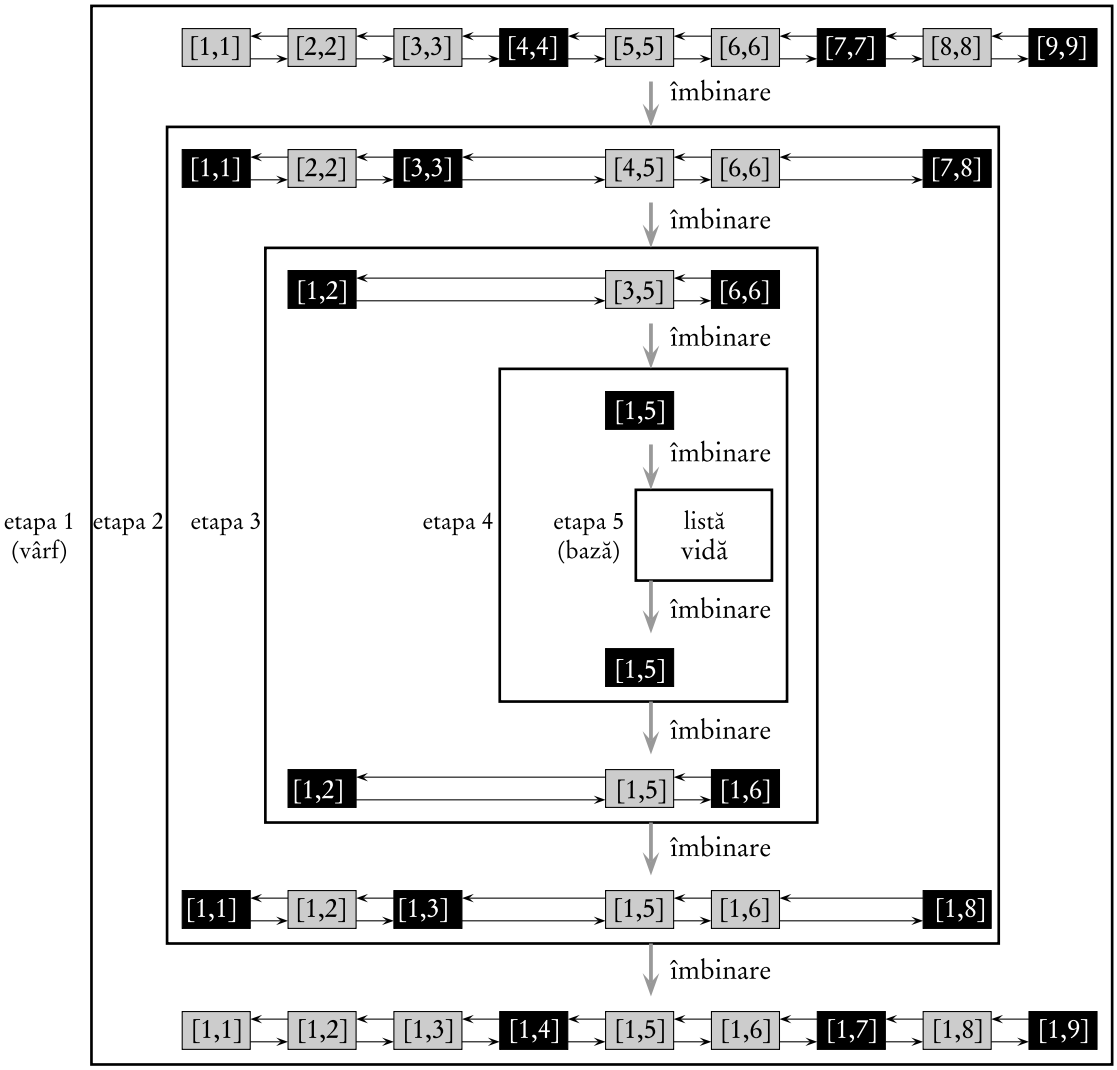


Figura 30.10 Etapele recursive ale algoritmului LISTARE-ALEATOARE-DE-PREFIX, prezentate pentru $n = 9$ obiecte originale. În fiecare etapă, obiectele desenate cu negru sunt eliminate. Procedura se autoapelează până când lista devine vidă, iar, apoi, obiectele eliminate sunt reintroduse în listă.

Din nefericire, această analiză simplă nu demonstrează că timpul aşteptat de execuţie al algoritmului LISTARE-ALEATOARE-DE-PREFIX este $\Theta(\lg n)$. De exemplu, dacă majoritatea procesoarelor îşi elimină obiectele lor repede şi câteva procesoare îşi elimină obiectele mult mai încet, timpul mediu în care un procesor îşi elimină obiectele poate să fie tot $\Theta(\lg n)$, dar timpul de execuţie al algoritmului ar putea fi mare.

Timpul de execuţie aşteptat al procedurii LISTARE-ALEATOARE-DE-PREFIX este, într-adevăr, $\Theta(\lg n)$, deşi analiza simplă de mai sus nu demonstrează riguros acest lucru. Vom folosi un argument cu probabilitate mare pentru a demonstra, cu o probabilitate de cel puţin $1 - 1/n$, că toate obiectele sunt eliminate în cel mult $c \lg n$ paşi recursivi, unde c este o constantă. Exerciţiile 30.4-4 şi 30.4-5 vă cer să generalizaţi această argumentaţie, pentru a demonstra limita de $\Theta(\lg n)$ a timpului de execuţie aşteptat.

Argumentul nostru de mare probabilitate se bazează pe observaţia că experimentul unui procesor dat de eliminare a obiectelor pe care le alege poate fi privit ca un şir de încercări Bernoulli (vezi capitolul 6). Experimentul are succes dacă obiectul este selectat pentru eliminare şi eşuează, în caz contrar. Deoarece suntem interesaţi să arătăm că probabilitatea de a obţine foarte puţine succese este mică, putem presupune că succesele apar cu o probabilitate de exact $1/4$, în loc de a apărea cu o probabilitate de cel puţin $1/4$. (Vezi exerciţiile 6.4-8 şi 6.4-9 pentru o justificare formală a unor presupuneri similare.)

Pentru a simplifica şi mai mult analiza, presupunem că există exact $n/\lg n$ procesoare, fiecare având $\lg n$ obiecte. Efectuăm $c \lg n$ încercări, unde c este o constantă ce va fi determinată ulterior, şi ne interesează evenimentul când apar mai puţin de $\lg n$ succese. Fie X variabila aleatoare care memorează numărul total de succese. Din corolarul 6.3, probabilitatea ca un procesor să elimine mai puţin de $\lg n$ obiecte în $c \lg n$ încercări este cel mult

$$\begin{aligned} Pr\{X < \lg n\} &\leq \binom{c \lg n}{\lg n} \left(\frac{3}{4}\right)^{c \lg n - \lg n} \leq \left(\frac{ec \lg n}{\lg n}\right)^{\lg n} \left(\frac{3}{4}\right)^{(c-1) \lg n} \\ &= \left(ec \left(\frac{3}{4}\right)^{c-1}\right)^{\lg n} \leq \left(\frac{1}{4}\right)^{\lg n} = 1/n^2, \end{aligned}$$

câtă vreme $c \geq 20$. (A doua linie rezultă din inegalitatea (6.9).) Astfel, probabilitatea ca obiectele aparţinând unui procesor dat să nu fie toate eliminate după $c \lg n$ paşi este de cel mult $1/n^2$.

Acum, dorim să determinăm o limită pentru probabilitatea că nu toate obiectele aparţinând tuturor procesoarelor au fost eliminate după $c \lg n$ paşi. Din inegalitatea lui Boole (6.22), această probabilitate este mai mică sau egală cu suma probabilităţilor că fiecare procesor nu şi-a eliminat obiectele. Deoarece există $n/\lg n$ procesoare şi fiecare procesor are o probabilitate de cel mult $1/n^2$ de a nu-şi elimina toate obiectele, probabilitatea ca un procesor oarecare să nu-şi fi terminat toate obiectele este cel mult

$$\frac{n}{\lg n} \cdot \frac{1}{n^2} \leq \frac{1}{n}$$

Am demonstrat astfel, cu o probabilitate de cel puţin $1 - 1/n$, că fiecare obiect este eliminat după $O(\lg n)$ apeluri recursive. Deoarece fiecare apel recusiv consumă un timp $O(1)$, LISTARE-ALEATOARE-DE-PREFIX se execută, cu o mare probabilitate, într-un timp $O(\lg n)$.

Constanta $c \geq 20$ din timpul de execuţie $c \lg n$ poate părea un pic prea mare pentru ca algoritmul să fie practic. De fapt, această constantă este mai mult un artefact al analizei decât o reflectare a performanţei algoritmului. În practică, algoritmul este, de obicei, rapid. Factorii

constanţi din analiză sunt mari deoarece evenimentul când un procesor îşi termină de eliminat toate obiectele sale din listă este dependent de evenimentul când un alt procesor îşi termină toate sarcinile. Din cauza acestor dependenţe, am folosit inegalitatea lui Boole, care nu necesită o relaţie de independenţă, dar are ca rezultat o constantă care este, în general, mai slabă decât cea întâlnită în practică.

Exerciţii

30.4-1 Ilustraţi prin desene ce erori pot să apară în execuţia algoritmului LISTARE-ALEATOARE-DE-PREFIX dacă două obiecte adiacente din listă sunt selectate pentru eliminare.

30.4-2 ⋆ Propuneţi o schimbare simplă pentru a face algoritmul LISTARE-ALEATOARE-DE-PREFIX să se execute, în cel mai defavorabil caz, într-un timp $O(n)$, pentru o listă de n obiecte. Folosiţi definiţia aşteptării pentru a demonstra că, având această modificare, algoritmul are un timp de execuţie aşteptat $O(\lg n)$.

30.4-3 ⋆ Arătaţi cum se poate implementa algoritmul LISTARE-ALEATOARE-DE-PREFIX, astfel încât să utilizeze cel mult $O(n/p)$ spaţiu pe procesor, în cel mai defavorabil caz, independent de cât de adâncă devine recursivitatea.

30.4-4 ⋆ Arătaţi că, pentru orice constantă $k \geq 1$, algoritmul LISTARE-ALEATOARE-DE-PREFIX rulează într-un timp $O(\lg n)$ cu o probabilitate de cel puţin $1 - 1/n^k$. Arătaţi cum influenţează k, constanta din limita timpului de execuţie.

30.4-5 ⋆ Folosind rezultatul din exerciţiul 30.4-4, arătaţi că timpul de execuţie aşteptat al algoritmului LISTARE-ALEATOARE-DE-PREFIX este $O(\lg n)$.

30.5. Întreruperi deterministe de simetrie

Să considerăm o situaţie în care două procesoare doresc ambele să obţină acces exclusiv la un obiect. Cum pot determina procesoarele care dintre ele va obţine primul accesul? Dorim să evităm atât cazul în care ambelor li se acordă accesul, cât şi cazul în care nu i se acordă accesul nici unuia. Problema alegerii unuia din procesoare este un exemplu de ***întrerupere de simetrie***. Am văzut cu toţii confuzia momentană şi impasul diplomatic care se ivesc atunci când două persoane încearcă să intre, simultan, pe o uşă. Probleme similare de întrerupere de simetrie sunt foarte răspândite în designul algoritmilor paraleli, iar soluţiile eficiente sunt extrem de folositoare.

O metodă de întrerupere a simetriei este aruncarea unei monezi. Pe un calculator, aruncarea monezii poate fi implementată cu ajutorul unui generator de numere aleatoare. Pentru exemplul cu cele două procesoare, ambele procesoare pot arunca monezi. Dacă unul obţine CAP şi celălalt BAN, cel care obţine CAP primeşte accesul. Dacă ambele procesoare obţin aceeaşi valoare, ele aruncă din nou monezile. Cu această strategie, simetria este întreruptă într-un timp aşteptat constant (vezi exerciţiul 30.5-1).

Am văzut efectivitatea unei strategii aleatoare în secţiunea 30.4. În algoritmul LISTARE-ALEATOARE-DE-PREFIX, obiectele adiacente din listă nu pot fi selectate pentru eliminare, dar

trebuie selectate cât mai multe obiecte din cele alese. Totuşi, într-o listă de obiecte alese, toate obiectele arată la fel. După cum am observat, folosirea numerelor aleatoare oferă un mijloc simplu şi eficient pentru a întrerupe simetria dintre obiectele adiacente din listă, garantând, de asemenea, cu o mare probabilitate, că multe obiecte sunt selectate.

În această secţiune, vom cerceta o metodă deterministă de întrerupere a simetriei. Cheia algoritmului este folosirea indicilor de procesor sau a adreselor de memorie în locul aruncării aleatoare de monezi. De exemplu, în cazul celor două procesoare, putem întrerupe simetria permiţând procesorului cu indicele de procesor mai mic să obţină primul accesul – un proces, evident, constant din punct de vedere al timpului.

Vom folosi aceeaşi idee, dar într-un mod mult mai inteligent, într-un algoritm pentru întreruperea simetriei într-o listă înlănţuită de n obiecte. Scopul este alegerea unei părţi constante dintre obiectele din listă, evitând însă alegerea a două obiecte adiacente. Algoritmul se execută, pe o maşină cu n procesoare, într-un timp $O(\lg^* n)$, fiind implementat într-o manieră EREW. Deoarece $\lg^* n \leq 5$ pentru orice $n \leq 2^{65536}$, valoarea $\lg^* n$ poate fi privită ca o constantă mică pentru orice scop practic (vezi pagina 32).

Algoritmul nostru determinist are două părţi. Prima parte determină o "6-colorare" a listei înlănţuite într-un timp $O(\lg^* n)$. A doua parte converteşte 6-colorarea într-o "mulţime independentă maximală" a listei, într-un timp $O(1)$. Mulţimea independentă maximală va conţine o parte constantă dintre cele n obiecte din listă şi nu vor exista două elemente în mulţime care să fie adicente.

Colorări şi mulţimi independente maximale

O ***colorare*** a unui graf neorientat $G = (V, E)$ este o funcţie $C : V \to \mathbb{N}$, astfel încât, oricare ar fi $u, v \in V$, dacă $C(u) = C(v)$, atunci $(u, v) \notin E$. Cu alte cuvinte, nu există vârfuri adiacente având aceeaşi culoare. Într-o 6-colorare a unei liste înlănţuite, toate culorile aparţin domeniului $\{0, 1, 2, 3, 4, 5\}$ şi nu există două vârfuri consecutive care să aibă aceeaşi culoare. De fapt, orice listă are o 2-colorare deoarece putem colora elementele având indicele impar cu 0 şi elementele având indicele par cu 1. Putem calcula o astfel de colorare într-un timp $O(\lg n)$ folosind un calcul paralel de prefix, dar, pentru multe aplicaţii, este suficient să calculăm o colorare într-un timp $O(1)$. Vom arăta acum că o 6-colorare poate fi calculată într-un timp $O(\lg^* n)$ fără a folosi numere aleatoare.

O ***mulţime independentă*** a unui graf $G = (V, E)$ este o submulţime de vârfuri $V' \subseteq V$, astfel încât orice muchie din E este incidentă cel mult unui vârf din V'. O ***mulţime independentă maximală*** este o mulţime independentă V' astfel încât, oricare ar fi un vârf $v \in V - V'$, mulţimea $V' \cup \{v\}$ nu este independentă – fiecare vârf care nu este în V' este adiacent unui vârf din V'. Nu confundaţi problema determinării unei mulţimi independente *maximale* – o problemă uşoară – cu problema determinării unei mulţimi independente *maxime* – o problemă dificilă. Problema determinării unei mulţimi independente de dimensiune maximă, într-un graf obişnuit, este NP-completă. (Vezi capitolul 36 pentru o discuţie despre NP-completitudine. Problema 36-1 se referă la mulţimi independente maxime.)

Pentru listele cu n obiecte, o mulţime independentă maximă (şi, deci, maximală) poate fi determinată într-un timp $O(\lg n)$ folosind calculul paralel de prefix, la fel ca în cazul 2-colorării, menţionate anterior, pentru a identifica obiectele cu rang impar. Această metodă selectează $\lceil n/2 \rceil$ obiecte. Observaţi, totuşi, că orice mulţime independentă maximală a unei liste înlănţuite conţine cel puţin $n/3$ obiecte deoarece, pentru oricare 3 obiecte consecutive, cel puţin unul trebuie să

facă parte din mulţime. Vom arăta, totuşi, că o mulţime maximală independentă a unei liste poate fi determinată într-un timp $O(1)$ dată fiind o colorare de timp $O(1)$ a listei.

Determinarea unei 6-colorări

Algoritmul ŞASE-CULORI calculează o 6-colorare a unei liste. Nu vom da un pseudocod pentru algoritm, dar îl vom descrie destul de detaliat. Vom presupune că, iniţial, fiecare obiect x din lista înlănţuită este asociat cu un procesor distinct $P(x) \in \{0, 1, \ldots, n-1\}$.

Ideea algoritmului ŞASE-CULORI este să calculăm un şir de culori $C_0[x], C_1[x], \ldots, C_m[x]$ pentru fiecare obiect x din listă. Colorarea iniţială C_0 este o ***n*-colorare**. Fiecare iteraţie a algoritmului defineşte o nouă colorare C_{k+1} bazată pe colorarea precedentă C_k, pentru $k = 0, 1, \ldots, m-1$. Colorarea finală C_m este o 6-colorare, şi vom demonstra că $m = O(\lg^* n)$.

Colorarea iniţială este o n-colorare trivială în care $C_0[x] = P(x)$. Deoarece în listă nu există două obiecte având aceeaşi culoare, nu există nici două obiecte adiacente în listă colorate la fel şi, deci, colorarea este corectă. Observaţi că fiecare dintre culorile iniţiale poate fi descrisă prin $\lceil \lg n \rceil$ biţi, ceea ce înseamnă că poate fi memorată într-un cuvânt de memorie.

Colorările următoare se obţin în modul următor. Iteraţia cu numărul de ordine k, pentru $k = 0, 1, \ldots, m-1$, începe cu o colorare C_k şi se sfârşeşte cu o colorare $C_k + 1$ ce foloseşte mai puţini biţi pentru un obiect, după cum arată prima parte a figurii 30.11. Să presupunem că, la începutul unei iteraţii, culoarea C_k a fiecărui obiect ocupă r biţi. Determinăm o nouă culoare a unui obiect x uitându-ne înainte în listă la culoarea lui $urm[x]$.

Pentru a fi mai exacţi, să presupunem că, pentru fiecare obiect x, avem $C_k[x] = a$ şi $C_k[urm[x]] = b$, unde $a = \langle a_{r-1}, a_{r-2}, \ldots, a_0 \rangle$ şi $b = \langle b_{r-1}, b_{r-2}, \ldots, b_0 \rangle$ sunt culori pe r biţi. Deoarece $C_k[x] \neq C_k[urm[x]]$, există un cel mai mic index i la care biţii celor două culori diferă: $a_i \neq b_i$. Deoarece $0 \leq i \leq r-1$, îl putem scrie pe i cu numai $\lceil \lg r \rceil$ biţi: $i = \langle i_{\lceil \lg r \rceil - 1}, i_{\lceil \lg r \rceil - 2}, \ldots, i_0 \rangle$. Recolorăm x cu valoarea lui i concatenată cu bitul a_i. Cu alte cuvinte, facem atribuirea

$$C_{k+1}[x] = \langle i, a_i \rangle = \langle i_{\lceil \lg r \rceil - 1}, i_{\lceil \lg r \rceil - 2}, \ldots, i_0, a_i \rangle .$$

Ultimul element din listă primeşte noua culoare $\langle 0, a_0 \rangle$. De aceea, numărul de biţi din fiecare nouă culoare este cel mult $\lceil \lg r \rceil + 1$.

Trebuie să demonstrăm că, dacă fiecare iteraţie a algoritmului ŞASE-CULORI porneşte cu o colorare, noua "colorare" pe care o produce este, într-adevăr, o colorare corectă. Pentru a face acest lucru, demonstrăm că $C_k[x] \neq C_k[urm[x]]$ implică $C_{k+1}[x] \neq C_{k+1}[urm[x]]$. Să presupunem că $C_k[x] = a$ şi $C_k[urm[x]] = b$, şi că $C_{k+1}[x] = \langle i, a_i \rangle$ şi $C_{k+1}[urm[x]] = \langle j, b_j \rangle$. Trebuie să considerăm două cazuri. Dacă $i \neq j$, atunci $\langle i, a_i \rangle \neq \langle j, b_j \rangle$, şi astfel culorile noi sunt diferite. Dacă $i = j$, atunci $a_i \neq b_i = b_j$ din cauza metodei noastre de recolorare, şi, astfel, noile culori sunt, încă o dată, diferite. (Situaţia din coada listei poate fi tratată similar).

Metoda de recolorare folosită de algoritmul ŞASE-CULORI ia o culoare de r biţi şi o înlocuieşte cu una de $(\lceil \lg r \rceil + 1)$ biţi, ceea ce înseamnă că numărul de biţi este strict redus pentru $r \geq 4$. Când $r = 3$, două culori pot diferi prin biţii din poziţiile 0, 1 sau 2. De aceea, fiecare culoare nouă este $\langle 00 \rangle$, $\langle 01 \rangle$ sau $\langle 10 \rangle$, concatenată fie cu 0, fie cu 1, rezultând, astfel, încă o dată, un număr de 3 biţi. Totuşi, numai 6 din cele 8 valori posibile pentru numerele de 3 biţi sunt folosite, astfel încât ŞASE-CULORI se termină într-adevăr cu o 6-colorare.

Presupunând că fiecare procesor poate determina indexul corespunzător i într-un timp $O(1)$ şi poate executa o operaţie de deplasare la stânga a biţilor într-un timp $O(1)$ – operaţii suportate în

mod obişnuit de multe maşini actuale – fiecare iteraţie se execută într-un timp $O(1)$. Algoritmul ŞASE-CULORI este un algoritm EREW: pentru fiecare obiect x, procesorul său accesează numai pe x şi pe $urm[x]$.

În final, să vedem de ce sunt necesare numai $O(\lg^* n)$ iteraţii pentru a aduce n-colorarea iniţială la o 6-colorare. Am definit $\lg^* n$ ca fiind numărul de aplicări ale funcţiei logaritm necesare pentru a-l reduce pe n la o valoare aproapiată de 1 sau, notând cu $\lg^{(i)} n$ i aplicări succesive ale funcţiei lg,

$$\lg^* n = \min\left\{i \geq 0 : \lg^{(i)} n \leq 1\right\}.$$

Fie r_i numărul de biţi din colorare chiar înainte de iteraţia cu numărul de ordine i. Vom demonstra prin inducţie că, dacă $\lceil \lg^{(i)} n \rceil \geq 2$, atunci $r_i \leq \lceil \lg^{(i)} n \rceil + 2$. Iniţial, avem $r_1 \leq \lceil \lg n \rceil$. Cea de-a i-a iteraţie scade numărul de biţi din colorare la $r_{i+1} = \lceil \lg r_i \rceil + 1$. Presupunând că ipoteza inductivă este adevărată pentru r_{i-1}, obţinem

$$\begin{aligned} r_i &= \lceil \lg r_{i-1} \rceil + 1 \leq \lceil \lg(\lceil \lg^{(i-1)} n \rceil + 2) \rceil + 1 \leq \lceil \lg(\lg^{(i-1)} n + 3) \rceil + 1 \\ &\leq \lceil \lg(2 \lg^{(i-1)} n) \rceil + 1 = \lceil \lg(\lg^{(i-1)} n) + 1 \rceil + 1 = \lceil \lg^{(i)} n \rceil + 2. \end{aligned}$$

A patra relaţie rezultă din presupunerea că $\lceil \lg^{(i)} n \rceil \geq 2$, ceea ce înseamnă că $\lceil \lg^{(i-1)} n \rceil \geq 3$. De aceea, după $m = \lg^* n$ paşi, numărul biţilor din colorare este $r_m \leq \lceil \lg^{(m)} n \rceil + 2 = 3$ deoarece $\lg^{(m)} n \leq 1$, din definiţia funcţiei $\lg^*$. Astfel, cel mult încă o iteraţie este de ajuns pentru a produce o 6-colorare. De aceea, timpul total de execuţie al algoritmului ŞASE-CULORI este $O(\lg^* n)$.

Determinarea unei mulţimi independente maximale dintr-o 6-colorare

Colorarea este partea dificilă a întreruperii simetriei. Algoritmul EREW LISTEAZĂ-MIM foloseşte n procesoare pentru a găsi o mulţime independentă maximală într-un timp $O(c)$, dată fiind o c-colorare a unei liste de n obiecte. Astfel, o dată ce am determinat o 6-colorare a unei liste, putem găsi o mulţime independentă maximală într-un timp $O(1)$.

Partea de la sfârşitul figurii 30.11 ilustrează ideea ce stă la baza algorimului LISTEAZĂ-MIM. Se dă o c-colorare C. Cu fiecare obiect x, păstrăm un bit $posibil[x]$, care ne spune dacă x este încă un candidat pentru includerea în mulţimea independentă maximală (MIM). Iniţial, $posibil[x]$ = ADEVĂRAT, pentru toate obiectele din x.

Apoi, algoritmul iterează fiecare din cele c culori. În iteraţia pentru culoarea i, fiecare procesor responsabil cu un obiect x verifică dacă $C[x] = i$ şi $posibil[x]$ = ADEVĂRAT. Dacă ambele condiţii sunt adevărate, atunci procesorul marchează x ca aparţinând mulţimii independente maximale ce este construită. Toate obiectele adiacente celor adăugate mulţimii independente maximale – acelea imediat înainte sau imediat după – vor avea biţii lor $posibil$ setaţi pe FALS. Ele nu pot aparţine mulţimii independente maximale pentru că sunt adiacente unui obiect ce se află în ea. După c iteraţii, fiecare obiect a fost fie "omorât" – bitul său $posibil$ a fost setat pe FALS, fie plasat în mulţimea maximală independentă.

Trebuie să arătăm că mulţimea rezultată este independentă şi maximală. Pentru a arăta că este independentă, să presupunem că două obiecte adiacente x şi $urm[x]$ sunt plasate în mulţime. Deoarece ele sunt adiacente, $C[x] \neq C[urm[x]]$ pentru că C este o colorare. Fără a pierde din generalitate, presupunem că $C[x] < C[urm[x]]$, astfel încât x este plasat în mulţime înaintea lui $urm[x]$. Dar în acest caz $posibil[urm[x]]$ a fost deja setat pe FALS atunci când obiectele de culoare $C[urm[x]]$ sunt luate în considerare şi, deci, $urm[x]$ nu poate să fie plasat în mulţime.

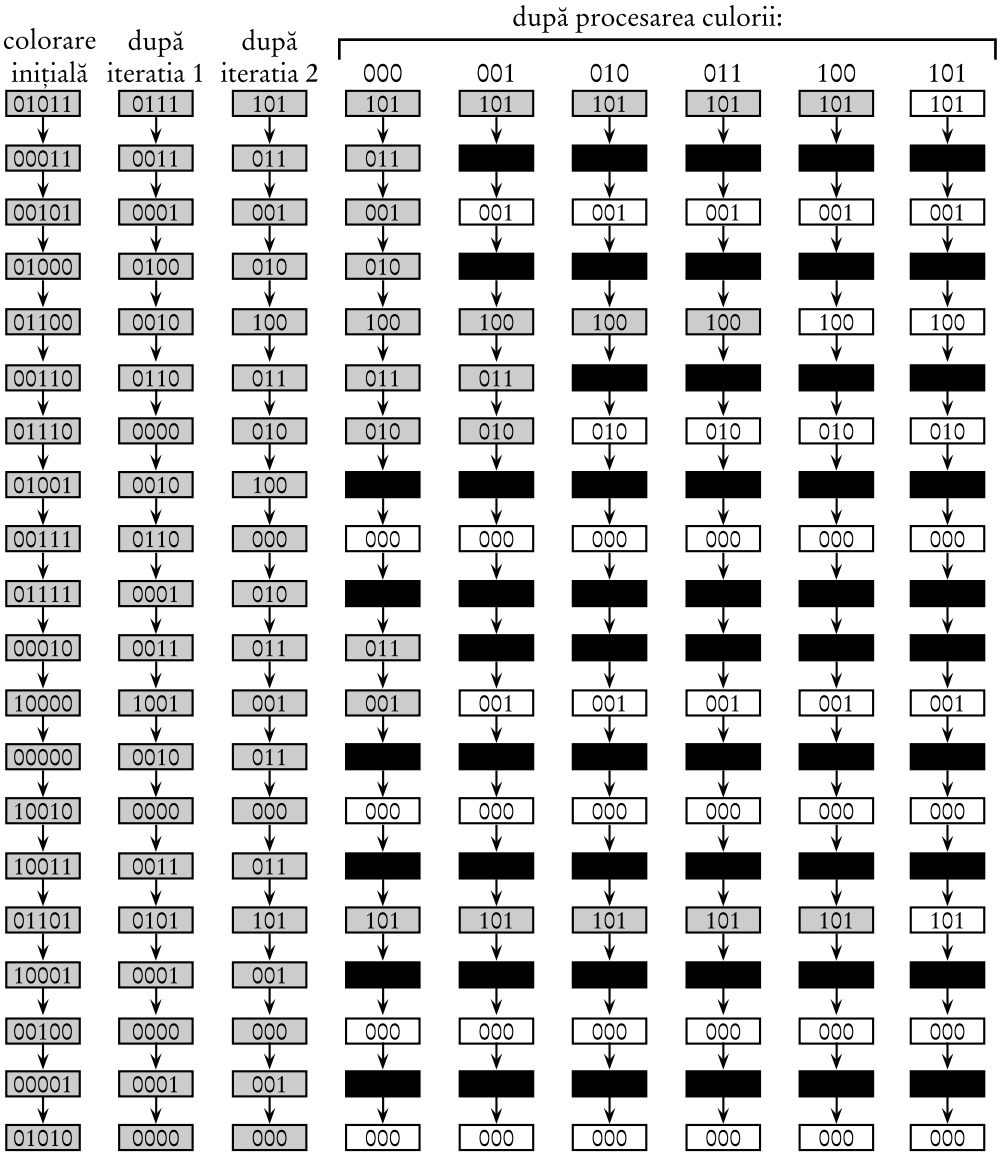


Figura 30.11 Algoritmii ŞASE-CULORI şi LISTEAZĂ-MIM care rup simetria dintr-o listă. Împreună, algoritmii determină o mulţime mare de obiecte neadiacente într-un timp $O(\lg^* n)$ folosind n procesoare. Lista iniţială de $n = 20$ obiecte este prezentată în partea stângă, pe verticală. Fiecare obiect are o culoare iniţială, distinctă, de 5 culori. Pentru aceşti parametri, algoritmul ŞASE-CULORI nu necesită decât cele două iteraţii prezentate pentru a recolora fiecare obiect cu o culoare din domeniul $\{0, 1, 2, 3, 4, 5\}$. Obiectele albe sunt plasate în mulţimea independentă maximală (MIM) de către algoritmul LISTEAZĂ-MIM pe măsură ce culorile sunt procesate, iar obiectele negre eliminate.

Pentru a arăta că mulţimea este maximală, să presupunem că nici unul dintre obiectele consecutive x, y si z nu a fost plasat în mulţime. Singurul mod în care y ar fi putut evita plasarea sa în mulţime este să fi fost eliminat atunci când un obiect adiacent a fost plasat în listă. Deoarece, din presupunerea noastră, nici x, nici z nu au fost plasate în mulţime, obiectul y trebuie să fi fost încă în viaţă în momentul când obiectele de culoare $C[y]$ au fost procesate. Deci, el trebuie să fi fost plasat în mulţime.

Fiecare iteraţie a algoritmului LISTEAZĂ-MIM necesită un timp $O(1)$ pe un PRAM. Algoritmul este EREW deoarece fiecare obiect se accesează doar pe sine însuşi, predecesorul şi succesorul său din listă. Combinând algoritmii LISTEAZĂ-MIM şi ŞASE-CULORI, putem întrerupe, deterministic, simetria dintr-o listă înlănţuită într-un timp $O(\lg^* n)$.

Exerciţii

30.5-1 Pentru exemplul de întrerupere de simetrie a două procesoare, prezentat la începutul acestei secţiuni, arătaţi că simetria este întreruptă într-un timp aşteptat constant.

30.5-2 Dată fiind o 6-colorare a unei liste de n obiecte, arătaţi cum se poate 3-colora lista într-un timp $O(1)$ folosind n procesoare pe un PRAM EREW.

30.5-3 Presupuneţi că fiecare nod dintr-un arbore cu n noduri, cu excepţia rădăcinii, are un pointer la părintele său. Daţi un algoritm CREW pentru a $O(1)$-colora arborele într-un timp $O(\lg^* n)$.

30.5-4 ⋆ Daţi un algoritm PRAM eficient pentru a $O(1)$-colora un graf de gradul 3. Analizaţi algoritmul propus.

30.5-5 O mulţime ***k-conducătoare*** a unei liste înlănţuite este o mulţime de obiecte (conducători) din listă, astfel încât nu există doi conducători adiacenţi şi, cel mult, k neconducători (subiecţi) separă conducătorii. Astfel, o mulţime independentă maximală este o mulţime 2-conducătoare. Arătaţi cum se poate calcula o mulţime $O(\lg n)$-conducătoare a unei liste de n obiecte într-un timp $O(1)$, folosind n procesoare. Arătaţi cum se poate calcula o mulţime $O(\lg \lg n)$-conducătoare într-un timp $O(1)$, presupunând aceleaşi lucruri ca mai sus.

30.5-6 ⋆ Arătaţi cum se poate determina o 6-colorare a unei liste înlănţuite de n obiecte într-un timp $O(\lg(\lg^* n))$. Presupuneţi că fiecare procesor poate memora un tablou precalculat de dimensiune $O(\lg n)$. (*Indicaţie:* În algoritmul ŞASE-CULORI, de câte valori depinde culoarea finală a unui obiect?)

Probleme

30-1 Prefix paralel segmentat

La fel ca un calcul de prefix obişnuit, un ***calcul de prefix segmentat*** este definit cu ajutorul unui operator binar, asociativ $\otimes$. Algoritmul primeşte un şir de intrare $x = \langle x_1, x_2, \ldots, x_n \rangle$ ale cărui elemente provin dintr-un domeniu S şi un şir ***segment*** $b = \langle b_1, b_2, \ldots, b_n \rangle$ ale cărui elemente provin din domeniul $\{0, 1\}$, cu $b_1 = 1$. Algoritmul produce un şir de ieşire $y = \langle y_1, y_2, \ldots, y_n \rangle$, cu elemente din domeniul S. Biţii lui b determină o partiţionare a lui x şi y în segmente. Un nou

$b =$	1	0	0	1	0	1	1	0	0	0	0	0	1	0
$x =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$y =$	1	3	6	4	9	6	7	15	24	34	45	57	13	27

Figura 30.12 Un calcul al prefixului paralel segmentat cu şirul segment b, şirul de intrare x şi şirul de ieşire y. Există 5 segmente.

segment începe oricând $b_i = 1$, iar cel curent continuă dacă $b_i = 0$. Calculul de prefix segmentat execută un calcul de prefix independent în cadrul fiecărui segment al lui x pentru a produce segmentul corespunzător al lui y. Figura 30.12 ilustrează un calcul de prefix segmentat folosind adunarea obişnuită.

a. Se defineşte operatorul $\widehat{\otimes}$ pe perechile ordonate (a, z), $(a', z') \in \{0, 1\} \times S$ în modul următor:

$$(a, z) \widehat{\otimes} (a', z') = \begin{cases} (a, z \otimes z') & \text{dacă } a' = 0, \\ (1, z') & \text{dacă } a' = 1. \end{cases}$$

Demonstraţi că $\widehat{\otimes}$ este asociativ.

b. Arătaţi cum se poate implementa, într-un timp $O(\lg n)$, orice calcul de prefix segmentat pe o listă de n elemente pe un PRAM CRCW cu $p = n$ procesoare.

c. Descrieţi un algoritm EREW de timp $O(k \lg n)$ pentru ordonarea unei liste de n numere de k biţi.

30-2 Algoritm de determinare a maximului, eficient ca număr de procesoare
Dorim să determinăm maximul a n numere pe un PRAM CRCW cu $p = n$ procesoare.

a. Demonstraţi că problema determinării maximului a $m \leq p/2$ numere se poate reduce la problema determinării a maximului a, cel mult, m^2/p numere într-un timp $O(1)$ pe un PRAM CRCW având p procesoare.

b. Dacă pornim cu $m = \lfloor p/2 \rfloor$ numere, câte numere rămân după k iteraţii ale algoritmului în partea (a)?

c. Arătaţi că problema determinării maximului a n numere poate fi rezolvată într-un timp $O(\lg \lg n)$ pe un PRAM CRCW cu $p = n$ procesoare.

30-3 Componente conexe
În această problemă, investigăm un algoritm CRCW arbitrar pentru determinarea componentelor conexe ale unui graf neorientat $G = (V, E)$ care foloseşte $|V + E|$ procesoare. Structura de date folosită este o pădure de mulţimi disjuncte (vezi secţiunea 22.3). Fiecare vârf $v \in V$ menţine un pointer $p[v]$ la un părinte. Iniţial, $p[v] = v$: pointerul vârfului v este el însuşi. La sfârşitul algorimului, pentru oricare două vârfuri $u, v \in V$, avem $p[u] = p[v]$ dacă şi numai dacă $u \rightsquigarrow v$ în G. În timpul algorimului, cei p pointeri formează o pădure de arbori ***de pointeri*** cu rădăcină. O ***stea*** este un arbore de pointeri în care $p[u] = p[v]$ pentru toate vârfurile u şi v din arbore.

Algoritmul pentru determinarea componentelor conexe presupune că fiecare muchie $(u, v) \in E$ apare de două ori: o dată ca (u, v) şi o dată ca (v, u). Algorimul foloseşte două operaţii de bază,

CÂRLIG şi SALT, şi o subrutină STEA care setează $stea[v]$ = ADEVĂRAT dacă v aparţine unei stele.

```
CÂRLIG(G)
1: STEA(G)
2: pentru fiecare muchie (u, v) ∈ E[G], în paralel execută
3:    dacă stea[u] şi p[u] > p[v] atunci
4:       p[p[u]] ← p[v]
5: STEA(G)
6: pentru fiecare muchie (u, v) ∈ E[G], în paralel execută
7:    dacă star[u] şi p[u] ≠ p[v] atunci
8:       p[p[u]] ← p[v]
```

```
SALT(G)
1: pentru fiecare v ∈ V[G], în paralel execută
2:    p[v] ← p[p[v]]
```

Algoritmul pentru determinarea componentelor conexe apelează iniţial procedura CÂRLIG, iar apoi apelează, în continuu, procedurile CÂRLIG, SALT, CÂRLIG, SALT şi aşa mai departe, până când nici un pointer nu mai este schimbat de către procedura SALT. (Observaţi că procedura CÂRLIG este apelată de două ori înaintea primului apel al procedurii SALT).

a. Descrieţi procedura STEA(G) în pseudocod.

b. Arătaţi că cei p pointeri formează, într-adevăr, arbori cu rădăcină, pointerul nodului rădăcină fiind chiar rădăcina însăşi. Arătaţi că, dacă u şi v se află în acelaşi arbore de pointeri, $p[u] = p[v]$ dacă şi numai dacă $u \rightsquigarrow v$ în G.

c. Demonstraţi că algoritmul este corect: el se termină, iar când se termină, $p[u] = p[v]$ dacă şi numai dacă $u \rightsquigarrow v$ în G.

Pentru a analiza algoritmul pentru componente conexe, vom examina o singură componentă conexă C, despre care presupunem că are cel puţin două vârfuri. Să presupunem că, la un moment dat în cursul execuţiei algoritmului, C este formată dintr-o mulţime $\{T_i\}$ de arbori de pointeri. Definim potenţialul C ca fiind

$$\Phi(C) = \sum_{T_i} \text{înălţime}(T_i)$$

Scopul analizei noastre este să demonstrăm că fiecare iteraţie de agăţări şi salturi micşorează $\Phi(C)$ cu un factor constant.

d. Demonstraţi că, după apelul iniţial al procedurii CÂRLIG, următoarele apeluri ale procedurii CÂRLIG nu-l măresc niciodată pe $\Phi(C)$.

f. Arătaţi că, după fiecare apel al procedurii CÂRLIG, cu excepţia celui iniţial, nici un arbore de pointeri nu este o stea, decât dacă arborele de pointeri conţine toate vârfurile din C.

g. Argumentaţi că, dacă C nu a fost compactizată într-o singură stea, atunci, după un apel al procedurii SALT, $\Phi(C)$ este, cel mult, $2/3$ din valoarea sa anterioară. Ilustraţi cazul cel mai defavorabil.

h. Demonstraţi că algoritmul determină toate componentele conexe ale lui G într-un timp $O(\lg V)$.

30-4 Transpunerea unei imagini raster

Un tampon de cadru pentru o imagine raster poate fi privit ca o matrice de biţi M de dimensiuni $p \times p$. Dispozitivul fizic de afişare a imaginii rastru determină o submatrice aflată în partea de sus, stânga de dimensiuni $n \times n$ a lui M vizibilă pe ecranul utilizatorului. Se foloseşte o operaţie BITBLT (engl. BLock Transfer of BITs – Transfer în Bloc al Biţilor) pentru a muta un dreptunghi de biţi dintr-o poziţie în alta. Mai exact, BITBLT$(r_1, c_1, r_2, c_2, nr, nc, *)$ setează

$$M[r_2 + i, c_2 + j] \leftarrow M[r_2 + i, c_2 + j] * M[r_1 + i, c_1 + j]$$

pentru $i = 0, 1, \ldots, nr - 1$ şi $j = 0, 1, \ldots, nc - 1$, unde $*$ este oricare din cele 16 funcţii booleene cu două intrări.

Ne interesează transpunerea imaginii ($M[i, j] \leftarrow M[j, i]$) în porţiunea vizibilă a tamponului de cadru. Costul copierii biţilor se presupune a fi mai mic decât acela al apelării primitivei BITBLT şi, astfel, suntem interesaţi să folosim cât mai puţine operaţii BITBLT posibil.

Arătaţi că orice imagine de pe ecran poate fi transpusă cu $O(\lg n)$ operaţii BITBLT. Presupuneţi că p este suficient mai mare decât n astfel încât porţiunile invizibile ale tamponnului de cadru oferă destul spaţiu de stocare pentru implementarea algoritmului. De cât spaţiu adiţional de stocare aveţi nevoie? (*Indicaţie:* Folosiţi o abordare divide-şi-stăpâneşte paralelă, în care unele operaţii BITBLT sunt efectuate prin operaţii ŞI booleene.)

Note bibliografice

Akl [9], Karp şi Ramachandran [118], şi Leighton [135] studiază algorimii paraleli pentru probleme combinaţionale. Arhitecturi variate de maşini paralele sunt descrise de Hwang şi Briggs [109] şi de Hwang şi DeGroot [110].

Teoria calculului paralel a început în anii 1940, când J. Von Neumann [38] a introdus un model restrâns de calcul paralel numit automat celular, care, în esenţă, este un tablou bidimensional de procesoare cu stări finite, interconectate în formă de reţea. Modelul PRAM a fost formalizat în 1978 de către Fortune şi Wyllie [73], deşi mulţi alţi autori au discutat înainte modele esenţial asemănătoare.

Salturile de pointeri au fost introduse de către Wyllie [204]. Studiul calculului paralel de prefix îşi are originea în lucrările lui Ofman [152], în contextul adunării carry-lookahead. Tehnica turului lui Euler se datorează lui Tarjan şi Vishkin [191].

Compromisuri în privinţa timpului procesor pentru determinarea maximului unei mulţimi de n numere au fost date de Valiant [193], care a demonstrat şi faptul că nu există un algoritm eficient de timp $O(1)$. Cook, Dwork şi Reischuk [50] au demonstrat că problema determinării maximului necesită un timp $\Omega(\lg n)$ pe un PRAM CRCW. Simularea unui algoritm CRCW printr-un algoritm EREW se datorează lui Vishkin [195].

Teorema 30.2 se datorează lui Brent [34]. Algoritmul aleator pentru determinarea eficientă a rangului listelor a fost descoperit de Anderson şi Miller [11]. Ei au dat, de asemenea, şi un algoritm deterministic eficient pentru aceeaşi problemă [10]. Algoritmul pentru întreruperea deterministică a simetriei se datorează lui Goldberg şi Plotkin [84]. El se bazează pe un algoritm similar, cu acelaşi timp de rulare, datorat lui Cole şi Vishkin [47].

31 Operaţii cu matrice

Operaţiile cu matrice sunt operaţii de bază în calcule ştiinţifice. De aceea, eficienţa algoritmilor asupra matricelor este de un interes considerabil. Acest capitol oferă o scurtă introducere în teoria matricelor, evidenţiind problema înmulţirii matricelor şi rezolvarea sistemelor de ecuaţii liniare.

După secţiunea 31.1, care introduce conceptele de bază şi notaţiile referitoare la matrice, secţiunea 31.2 prezintă algoritmul surprinzător al lui Strassen pentru înmulţirea a două matrice de ordinul $n \times n$ într-un timp de ordinul lui $\Theta(n^{\lg 7}) = O(n^{2.81})$. Secţiunea 31.3 defineşte noţiunile de cvasiinel, inel şi câmpuri clarificând ipotezele cerute de funcţionarea algoritmului lui Strassen. De asemenea, secţiunea conţine un algoritm asimptotic rapid pentru înmulţirea matricelor booleene. În secţiunea 31.4 se tratează modul de rezolvare a unui sistem de ecuaţii liniare utilizând descompunerea LUP. Apoi, în secţiunea 31.5 se studiază legătura strânsă dintre problemele înmulţirii matricelor şi problema inversării lor. În final, în secţiunea 31.6 se studiază clasa importantă a matricelor simetrice pozitiv-definite şi se prezintă modul lor de utilizare în rezolvarea unui sistem de n ecuaţii liniare cu m necunoscute $m < n$, folosind metoda celor mai mici pătrate.

31.1. Proprietăţile matricelor

În această secţiune, vom recapitula câteva concepte de bază din teoria matricelor precum şi proprietăţi fundamentale ale matricelor, concentrându-ne atenţia asupra acelora care vor fi necesare în secţiunile următoare.

Matrice şi vectori

O ***matrice*** este un tablou dreptunghiular de numere. De exemplu,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \tag{31.1}$$

este o matrice $A = (a_{ij})$ de ordinul 2×3, unde $i = 1, 2$ şi $j = 1, 2, 3$, iar elementul matricei din linia i şi coloana j este a_{ij}. Vom nota matricele cu litere mari, iar elementele lor cu litere mici, indexate. Mulţimea tuturor matricelor de ordinul $m \times n$, având elemente reale, se notează cu $\mathbb{R}^{m \times n}$. În general, mulţimea matricelor de ordinul $m \times n$, având elemente din mulţimea S, se notează cu $S^{m \times n}$.

Transpusa unei matrice A este matricea A^{T} care se obţine interschimbând liniile şi coloanele lui A. Pentru matricea A din relaţia (31.1),

$$A^{\mathrm{T}} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}.$$

Un ***vector*** este un tablou unidimensional de numere. De exemplu,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} . \tag{31.2}$$

este un vector de dimensiunea 3. Vom utiliza litere mici pentru a nota vectori şi vom nota prin x_i, pentru $i = 1, 2, \ldots, n$, al i-lea element al vectorului x de dimensiune n. Forma standard a unui vector este ***vectorul coloană***, echivalent cu o matrice de ordinul $n \times 1$; ***vectorul linie*** corespunzător se obţine luând transpusa:

$$x^{\mathrm{T}} = (2\ 3\ 5).$$

Vectorul unitate e_i este vectorul pentru care al i-lea element este 1 şi toate celelalte elemente sunt 0. De obicei, dimensiunea unui vector unitate rezultă din context.

O ***matrice zero*** este o matrice ale cărei elemente sunt 0. O astfel de matrice, de obicei, se notează cu 0, întrucât ambiguitatea dintre numărul 0 şi matricea 0 se rezolvă simplu, subînţelegându-se din context la ce se face referirea. Dacă este vorba de o matrice 0, atunci ordinul ei trebuie să rezulte din context.

Matricele ***pătratice*** de ordinul $n \times n$ sunt întâlnite frecvent. O importanţă mare prezintă diferitele cazuri particulare de matrice pătratice:

1. O ***matrice diagonală*** are elementele $a_{ij} = 0$ pentru $i \neq j$. Deoarece toate elementele nesituate pe diagonală sunt nule, matricea poate fi specificată enumerând elementele de pe diagonală:

$$\mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

2. ***Matricea unitate*** I_n de ordinul $n \times n$ este matricea având 1 pe diagonală:

$$I_n = \mathrm{diag}(1, 1, \ldots, 1) = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix}.$$

   În cazul în care I nu are indice, ordinul ei poate fi dedus din context. Coloana a i-a a unei matrice unitate este vectorul unitate e_i.

3. Numim ***matrice tridiagonală*** acea matrice T în care $t_{ij} = 0$ dacă $|i - j| > 1$. Elementele nenule apar numai pe diagonala principală, imediat deasupra diagonalei principale ($t_{i,i+1}$ pentru $i = 1, 2, \ldots, n - 1$) sau imediat sub diagonala principală ($t_{i+1,i}$, pentru $i =$

1, 2, ..., $n-1$):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \dots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \dots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \dots & t_{n-1,n-2} & t_{n-1,n-1} & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix}.$$

4. Numim ***matrice superior triunghiulară*** acea matrice U în care $u_{ij} = 0$ pentru orice $i > j$. Toate elementele de sub diagonală sunt nule:

$$U = \begin{pmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{21} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{pmatrix}.$$

O matrice superior triunghiulară este ***matrice unitate superior triunghiulară*** dacă toate elementele de pe diagonală sunt egale cu 1.

5. Numim ***matrice inferior triunghiulară*** acea matrice L în care $l_{ij} = 0$ pentru orice $i < j$. Toate elementele de deasupra diagonalei sunt nule:

$$L = \begin{pmatrix} l_{11} & 0 & \dots & 0 \\ l_{21} & l_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{pmatrix}.$$

O matrice inferior triunghiulată care are toate elementele de pe diagonală egale cu 1 se numeşte ***matrice unitate inferior triunghiulară***.

6. O ***matrice de permutare*** P este acea matrice care are exact un 1 în fiecare linie sau coloană şi zero în rest. Un exemplu de matrice de permutare este:

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}.$$

O astfel de matrice se numeşte matrice de permutare deoarece, înmulţind un vector x cu o matrice de permutare, se permută (rearanjează) elementele lui x.

7. O ***matrice simetrică*** A este o matrice care satisface condiţia $A = A^{\mathrm{T}}$. De exemplu,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

este o matrice simetrică.

Operaţii cu matrice

Elementele unei matrice sau ale unui vector sunt numere dintr-o clasă, cum ar fi numerele reale, complexe sau numere întregi modulo un număr prim. Clasa defineşte modul de adunare şi înmulţire a numerelor. Aceste definiţii se pot extinde pentru a include adunarea şi înmulţirea matricelor.

Definim ***adunarea matricelor*** după cum urmează. Dacă $A = (a_{ij})$ şi $B = (b_{ij})$ sunt matrice de ordinul $m \times n$, atunci matricea sumă $C = (c_{ij}) = A + B$ este matricea de ordinul $m \times n$ definită prin

$$c_{ij} = a_{ij} + b_{ij}$$

pentru $i = 1, 2, \ldots, m$ şi $j = 1, 2, \ldots, n$. După cum se observă, adunarea se face pe componente. Matricea zero este matricea unitate pentru adunarea matricelor:

$$A + 0 = A = 0 + A.$$

Dacă λ este un număr şi $A = (a_{ij})$ este o matrice, atunci $\lambda A = (\lambda a_{ij})$ este ***produsul scalar*** al lui A obţinut înmulţind fiecare din elementele sale cu λ. Ca un caz special, definim ***opusul*** unei matrice $A = (a_{ij})$ ca fiind $-1 \cdot A = -A$, deci elementul de indice ij al lui $-A$ este $-a_{ij}$. Astfel,

$$A + (-A) = 0 = (-A) + A.$$

Dându-se aceste definiţii, putem defini ***scăderea matricelor*** prin adunarea opusei unei matrice: $A - B = A + (-B)$. Definim ***înmulţirea matricelor*** după cum urmează. Pornim cu două matrice A şi B care sunt ***compatibile***, în sensul că numărul de coloane al matricei A este egal cu numărul de linii al matricei B. (În general, o expresie care conţine un produs de matrice AB implică faptul că matricele A şi B sunt compatibile.) Dacă $A = (a_{ij})$ este o matrice de ordinul $m \times n$ şi $B = (b_{jk})$ este o matrice de ordinul $n \times p$, atunci matricea produs $C = AB$ este o matrice $C = (c_{ik})$ de ordinul $m \times p$, unde:

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \tag{31.3}$$

pentru $i = 1, 2, \ldots, m$ şi $k = 1, 2, \ldots, p$. Procedura ÎNMULŢEŞTE-MATRICE din secţiunea 26.1 implementează înmulţirea matricelor într-o manieră directă bazată pe relaţia (31.3), presupunând că matricele sunt pătratice: $m = n = p$. Pentru a înmulţi matrice de ordinul $n \times n$, algoritmul ÎNMULŢEŞTE-MATRICE efectuează n^3 înmulţiri şi $n^2(n-1)$ adunări, timpul de execuţie fiind $\Theta(n^3)$.

Matricele au multe (dar nu toate) din proprietăţile tipice numerelor. Matricele unitate sunt unităţi pentru înmulţirea matricelor:

$$I_m A = A I_n = A$$

pentru orice matrice A de ordinul $m \times n$. Înmulţirea cu matricea zero dă o matrice zero:

$$A0 = 0.$$

Înmulţirea matricelor este asociativă:

$$A(BC) = (AB)C \tag{31.4}$$

pentru matricele compatibile A, B şi C.

Înmulţirea matricelor este distributivă în raport cu adunarea:

$$\begin{aligned} A(B+C) &= AB+AC, \\ (B+C)D &= BD+CD. \end{aligned} \tag{31.5}$$

Înmulţirea matricelor de ordinul $n \times n$ nu este comutativă decât dacă $n = 1$. De exemplu, dacă $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ şi $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$, atunci $AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ şi $BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$.

Pentru a calcula produse de forma matrice-vector sau vector-vector, se consideră că vectorul ar fi echivalent cu o matrice de ordinul $n \times 1$ (sau o matrice de ordinul $1 \times n$, în cazul unui vector linie). Astfel, dacă A este o matrice de ordinul $m \times n$ şi x un vector de dimensiune n, atunci Ax este un vector de dimensiune m. Dacă x şi y sunt vectori de dimensiune n, atunci

$$x^{\mathrm{T}} y = \sum_{i=1}^{n} x_i y_i$$

este un număr (de fapt, o matrice de ordinul 1×1) numit ***produs scalar*** al vectorilor x şi y. Matricea xy^{T} este o matrice Z de ordinul $n \times n$ numită ***produsul exterior*** al vectorilor x şi y, cu $z_{ij} = x_i y_j$. ***Norma (euclidiană)*** $\|x\|$ a vectorului x de dimensiune n se defineşte prin

$$\|x\| \;=\; (x_1^2 + x_2^2 + \ldots + x_n^2)^{1/2} \;=\; (x^{\mathrm{T}} x)^{1/2}.$$

Astfel, norma lui x este lungimea în spaţiul euclidian n-dimensional.

Inversul, rangul şi determinantul matricelor

Inversa unei matrice A de ordinul $n \times n$ este o matrice de ordinul $n \times n$, notată cu A^{-1} (dacă există), astfel încât $AA^{-1} = I_n = A^{-1}A$. De exemplu,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

Multe matrice de ordinul $n \times n$ nenule nu au inversă. O matrice care nu are inversă se numeşte ***neinversabilă*** sau ***singulară***. Un exemplu de matrice singulară nenulă este

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

Dacă o matrice are o inversă, ea se numeşte ***inversabilă*** sau ***nesingulară***. Inversele matricelor, dacă ele există, sunt unice. (Vezi exerciţiul 31.1-4.) Dacă A şi B sunt matrice de ordinul $n \times n$ nesingulare, atunci

$$(BA)^{-1} = A^{-1}B^{-1}. \tag{31.6}$$

Operaţia de inversare comută cu operaţia de transpunere:

$$(A^{-1})^{\mathrm{T}} = (A^{\mathrm{T}})^{-1}.$$

Vectorii $x_1, x_2, \ldots, x_n$ sunt ***liniar dependenţi*** dacă există coeficienţii $c_1, c_2, \ldots, c_n$, nu toţi nuli, astfel încât $c_1 x_1 + c_2 x_2 + \cdots + c_n x_n = 0$. De exemplu, vectorii $x_1 = (1\ 2\ 3)^{\mathrm{T}}, x_2 = (2\ 6\ 4)^{\mathrm{T}}$

şi $x_3 = (4\ 11\ 9)^{\mathrm{T}}$ sunt liniar dependenţi deoarece $2x_1 + 3x_2 - 2x_3 = 0$. Dacă vectorii nu sunt liniar dependenţi, atunci sunt ***liniar independenţi***. De exemplu, coloanele matricei unitate sunt liniar independente.

Rangul coloană al unei matrice A de ordinul $m \times n$ nenule este dimensiunea celei mai mari mulţimi de coloane liniar independente ale lui A. Similar, ***rangul linie*** al lui A este dimensiunea celei mai mari mulţimi de linii liniar independente ale lui A. O proprietate fundamentală a oricărei matrice A este faptul că rangurile linie şi coloană ale ei sunt totdeauna egale, aşa că putem, pur şi simplu, să ne referim la ***rangul*** lui A. Rangul unei matrice de ordinul $m \times n$ este un întreg între 0 şi $\min(m, n)$, inclusiv. (Rangul unei matrice zero este 0, iar al matricei unitate de ordinul $n \times n$ este n.) O definiţie echivalentă pentru rang, adesea mai utilă, este aceea că rangul unei matrice A de ordinul $m \times n$ nenule este cel mai mic număr r, astfel încât să existe matricele B şi C de ordine $m \times r$, respectiv, $r \times n$, astfel ca

$$A = BC.$$

O matrice pătrată de ordinul $n \times n$ are ***rangul complet*** dacă acesta este n. O proprietate fundamentală a rangurilor va fi enunţată prin următoarea teoremă.

Teorema 31.1 O matrice pătrată are rangul complet dacă şi numai dacă este nesingulară. ∎

O matrice de ordinul $m \times n$ are ***rangul coloană complet*** dacă rangul ei este n.

Un ***vector de anulare*** pentru o matrice A este un vector x diferit de zero, astfel încât $Ax = 0$. Teorema următoare, a cărei demonstraţie este cerută în exerciţiul 31.1-8, precum şi corolarul ei leagă noţiunile de rang coloană şi singularitate cu cea de vector de anulare.

Teorema 31.2 O matrice A are un rang coloană complet dacă şi numai dacă nu are un vector de anulare. ∎

Corolarul 31.3 O matrice pătratică A este singulară dacă şi numai dacă are un vector de anulare. ∎

Al ij-lea ***minor*** al matricei A de ordinul $n \times n$, pentru $n > 1$, este matricea $A_{[ij]}$ de ordinul $(n-1) \times (n-1)$, obţinută eliminând linia a i-a şi coloana a j-a din A. ***Determinantul*** unei matrice A de ordinul $n \times n$ poate fi definit recursiv în termenii minorilor ei, prin

$$\det(A) = \begin{cases} a_{11} & \text{dacă } n = 1, \\ a_{11}\det(A_{[11]}) - a_{12}\det(A_{[12]}) & \\ \quad + \cdots + (-1^{n+1} a_{1n}\det(A_{[1n]}) & \text{dacă } n > 1. \end{cases} \tag{31.7}$$

Termenul $(-1)^{i+j}\det(A_{[ij]})$ este cunoscut sub numele de ***complement algebric*** al elementului a_{ij}.

Următoarele teoreme, ale căror demonstraţii sunt omise aici, exprimă proprietăţile fundamentale ale determinanţilor.

Teorema 31.4 (Proprietăţile determinanţilor) Determinantul unei matrice pătratice A are următoarele proprietăţi:

- Dacă o linie sau o coloană oarecare a lui A este zero, atunci $\det(A)$=0.
- Dacă se înmulţesc elementele unei linii (sau unei coloane) cu λ, atunci determinantul lui A se înmulţeşte cu λ.

- Dacă elementele unei linii (sau coloane) se adaugă la elementele altei linii (sau coloane) valoarea determinantului nu se schimbă.
- Determinantul lui A este egal cu determinantul lui A^{T}.
- Dacă se permută între ele două linii (sau coloane), valoarea determinantului se înmulţeşte cu -1.

De asemenea, pentru orice matrice pătratice A şi B, avem $\det(AB) = \det(A)\det(B)$. ∎

Teorema 31.5 O matrice A de ordinul $n \times n$ este singulară dacă şi numai dacă $\det(A) = 0$. ∎

Matrice pozitiv-definite

Matricele pozitiv-definite joacă un rol important în multe aplicaţii. O matrice A de ordinul $n \times n$ este ***pozitiv-definită*** dacă $x^{\mathrm{T}}Ax > 0$ pentru orice vector $x \neq 0$, n-dimensional. De exemplu, matricea unitate este pozitiv-definită, deoarece pentru orice vector nenul $x = (x_1 \;\; x_2 \;\; \cdots \;\; x_n)^{\mathrm{T}}$,

$$x^{\mathrm{T}} I_n x \;=\; x^{\mathrm{T}} x \;=\; \|x\|^2 \;=\; \sum_{i=1}^{n} x_i^2 \;>\; 0.$$

Aşa cum vom vedea, conform următoarei teoreme, matricele care apar în aplicaţii sunt adesea pozitiv-definite.

Teorema 31.6 Pentru orice matrice A având rangul coloană complet, matricea $A^{\mathrm{T}}A$ este pozitiv definită.

Demonstraţie. Trebuie să arătăm că $x^{\mathrm{T}}(A^{\mathrm{T}}A)x > 0$ pentru orice vector x nenul. Pentru orice vector x, pe baza exerciţiului 31.1-3, avem

$$x^{\mathrm{T}}(A^{\mathrm{T}}A)x \;=\; (Ax)^{\mathrm{T}}(Ax) \;=\; \|Ax\|^2 \;\geq\; 0. \tag{31.8}$$

Să observăm că $\|Ax\|^2$ este chiar suma pătratelor elementelor vectorului Ax. De aceea, dacă $\|Ax\|^2 = 0$, fiecare element al lui Ax este 0, ceea ce înseamnă că $Ax = 0$. Întrucât A are rangul coloană complet, $Ax = 0$ implică $x = 0$ conform teoremei 31.2. Deci, $A^{\mathrm{T}}A$ este pozitiv-definită. ∎

Alte proprietăţi ale matricelor pozitiv-definite vor fi studiate în secţiunea 31.6.

Exerciţii

31.1-1 Demonstraţi că produsul a două matrice inferior triunghiulare este o matrice inferior triunghiulară. Demonstraţi că determinantul unei matrice triunghiulare (inferioară sau superioară) este egal cu produsul elementelor de pe diagonala ei. Demonstraţi că dacă o matrice inferior triunghiulară admite inversă, aceasta este o matrice inferior triunghiulară.

31.1-2 Demonstraţi că, dacă P este o matrice de permutare de ordinul $n \times n$ şi A este o matrice de ordinul $n \times n$, atunci PA poate fi obţinută din A permutând liniile sale, iar AP poate fi obţinută din A prin permutarea coloanelor sale. Demonstraţi că produsul a două matrice de permutare este o matrice de permutare. Demonstraţi că dacă P este o matrice de permutare, atunci este inversabilă, inversa ei este P^{T}, iar P^{T} este o matrice de permutare.

31.1-3 Demonstraţi că $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$ şi că $A^{\mathrm{T}}A$ este, întotdeauna, o matrice simetrică.

31.1-4 Demonstraţi că, dacă B şi C sunt inversele matricei A, atunci $B = C$.

31.1-5 Fie A şi B două matrice de ordinul $n \times n$, astfel încât $AB = I$. Arătaţi că, dacă A' este obţinută din A adăugând linia j la linia i, atunci inversa B' a lui A' poate fi obţinută scăzând coloana i din coloana j a matricei B.

31.1-6 Fie A o matrice de ordinul $n \times n$ nesingulară. Arătaţi că orice element din A^{-1} este real dacă şi numai dacă orice element din A este real.

31.1-7 Arătaţi că, dacă A este o matrice nesingulară simetrică, atunci A^{-1} este simetrică. Arătaţi că, dacă B este o matrice arbitrară (compatibilă), atunci BAB^{T} este simetrică.

31.1-8 Arătaţi că o matrice A are rangul coloană complet dacă şi numai dacă $Ax = 0$ implică $x = 0$. (*Indicaţie:* se va exprima dependenţa liniară a unei coloane de celelalte, sub forma unei ecuaţii matrice-vector.)

31.1-9 Demonstraţi că, pentru orice două matrice A şi B compatibile,

$$\mathrm{rang}(AB) \leq \min(\mathrm{rang}(A), \mathrm{rang}(B))$$

unde egalitatea are loc dacă A sau B este matrice pătratică nesingulară.(*Indicaţie:* utilizaţi definiţia alternativă a rangului unei matrice.)

31.1-10 Dându-se numerele $x_0, x_1, \ldots, x_{n-1}$, demonstraţi că determinantul ***matricei Vandermonde***

$$V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}$$

este

$$\det\left(V(x_0, x_1, \ldots, x_{n-1})\right) = \prod_{0 \leq j < k \leq n-1} (x_k - x_j).$$

(*Indicaţie:* Se înmulţeşte coloana i cu $-x_0$ şi se adună la coloana $i + 1$ pentru $i = n - 1,\ n - 2,\ \ldots\ 1$, apoi se utilizează inducţia.)

31.2. Algoritmul lui Strassen pentru înmulţirea matricelor

Această secţiune prezintă remarcabilul algoritm recursiv al lui Strassen de înmulţire a matricelor de ordinul $n \times n$, care se execută într-un timp de ordin $\Theta(n^{\lg 7}) = O(n^{2.81})$. Astfel, pentru n suficient de mare, algoritmul este mai performant decât algoritmul ÎNMULŢEŞTE-MATRICE, prezentat în secţiunea 26.1, pentru care timpul este de ordinul $\Theta(n^3)$.

Prezentarea generală a algoritmului

Algoritmul lui Strassen poate fi privit ca o aplicaţie a unei tehnici familiare de proiectare: divide şi stăpâneşte. Să presupunem că dorim să calculăm produsul $C = AB$, unde fiecare dintre A, B şi C sunt matrice de ordinul $n \times n$. Presupunând că n este multiplu de 2, împărţim fiecare din matricele A, B şi C în $n/2 \times n/2$ matrice, rescriind relaţia $C = AB$ astfel:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \tag{31.9}$$

(Exerciţiul 31.2-2 tratează situaţiile în care n nu este multiplu de 2.) Pentru ca notaţiile să fie convenabile, elementele lui A se notează alfabetic de la stânga la dreapta, în timp ce elementele lui B se notează de sus în jos, în concordanţă cu modul de realizare a înmulţirii matricelor. Relaţia (31.9) corespunde următoarelor patru relaţii

$$r = ae + bf, \tag{31.10}$$

$$s = ag + bh, \tag{31.11}$$

$$t = ce + df, \tag{31.12}$$

$$u = cg + dh. \tag{31.13}$$

Fiecare dintre aceste patru relaţii specifică două înmulţiri de matrice de ordinul $n/2 \times n/2$ şi suma lor. Utilizând aceste relaţii pentru a defini o strategie directă a tehnicii divide şi stăpâneşte, deducem următoarea relaţie de recurenţă pentru timpul $T(n)$ de înmulţire a două matrice de ordinul $n \times n$:

$$T(n) = 8T(n/2) + \Theta(n^2). \tag{31.14}$$

Din nefericire, relaţia (31.14) are soluţia $T(n) = \Theta(n^3)$ şi, deci, această metodă nu este mai rapidă decât cea obişnuită.

Strassen a descoperit o metodă recursivă diferită care cere numai şapte înmulţiri recursive de matrice de ordinul $n/2 \times n/2$ şi $\Theta(n^2)$ adunări şi scăderi scalare, obţinându-se relaţia de recurenţă

$$T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\lg 7}) = O(n^{2.81}). \tag{31.15}$$

Metoda lui Strassen are patru paşi:

1. Se descompun matricele A şi B date în submatrice de ordinul $n/2 \times n/2$, ca în relaţia (31.9).
2. Utilizând $\Theta(n^2)$ adunări şi scăderi scalare, se calculează 14 matrice de ordinul $n/2 \times n/2$: $A_1,\ B_1,\ A_2,\ B_2,\ \cdots,\ A_7,\ B_7$.
3. Se calculează recursiv 7 matrice produs: $P_i = A_i B_i$ pentru $i = 1,\ 2,\ \ldots,\ 7$.
4. Se calculează submatricele $r,\ s,\ t,\ u$ ale matricei rezultat C prin adunarea şi/sau scăderea diferitelor combinaţii ale matricelor P_i, utilizând numai $\Theta(n^2)$ adunări şi scăderi scalare.

O astfel de procedură satisface relaţia de recurenţă (31.15). Tot ce trebuie să facem în continuare este să completăm detaliile absente.

Determinarea produselor de submatrice

Modul în care Strassen a descoperit submatricele ce reprezintă cheia acestui algoritm nu este bine cunoscut. Reconstruim o metodă plauzibilă a determinării submatricelor.

Să presupunem că am ghicit că fiecare matrice P_i poate fi pusă sub forma

$$P_i = A_i B_i = (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h), \tag{31.16}$$

unde coeficienţii α_{ij}, β_{ij} aparţin mulţimii $\{-1, 0, 1\}$. Adică, am ghicit că fiecare produs se calculează adunând sau scăzând nişte submatrice din A, adunând sau scăzând nişte submatrice din B şi înmulţind cele două rezultate. Deşi sunt posibile strategii mai generale, această strategie se dovedeşte a fi funcţională.

Dacă formăm toate cele patru produse în acest fel, atunci, putem utiliza această metodă recursiv, fără a presupune comutativitatea înmulţirii deoarece fiecare produs are toate submatricele lui A în stânga şi toate submatricele lui B în dreapta. Această proprietate este esenţială pentru aplicarea recursivă a acestei metode deoarece înmulţirea matricelor nu este comutativă.

Este convenabil să utilizăm o matrice de ordinul 4×4 pentru a reprezenta combinaţiile liniare ale produselor de submatrice, unde fiecare produs combină o submatrice a lui A cu o submatrice a lui B, ca în relaţia (31.16). De asemenea, ecuaţia (31.10) poate fi scrisă astfel

$$r = ae + bf = \begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{matrix} \\ a \\ b \\ c \\ d \end{matrix} \begin{matrix} \begin{matrix} e & f & g & h \end{matrix} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{matrix}.$$

Ultima expresie utilizează o notaţie prescurtată în care "+" reprezintă $+1$, "." reprezintă 0, iar "$-$" reprezintă -1. (În continuare, vom omite etichetele pentru linii şi coloane.) Utilizând această notaţie, obţinem următoarele relaţii pentru celelalte submatrice ale matricei rezultat C:

$$s = ag + bh = \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

$$t = ce + df = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix},$$

$$u = cg + dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Vom căuta un algoritm mai rapid de înmulţire de matrice observând că submatricea s poate fi calculată prin relaţia $s = P_1 + P_2$, unde P_1 şi P_2 se calculează utilizând o înmulţire de matrice:

$$P_1 = A_1 B_1 = a \cdot (g - h) = ag - ah = \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$P_2 \;=\; A_2B_2 \;=\; (a+b)\cdot h \;=\; ah+bh \;=\; \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Matricea t poate fi calculată într-o manieră similară prin relaţia $t = P_3 + P_4$, unde

$$P_3 \;=\; A_3B_3 \;=\; (c+d)\cdot e \;=\; ce+de \;=\; \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}$$

şi

$$P_4 \;=\; A_4B_4 \;=\; d\cdot(f-e) \;=\; df-de \;=\; \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}.$$

Numim ***termen principal*** unul din cei opt termeni care apar în părţile drepte ale uneia din relaţiile (31.10)–(31.13). Am utilizat acum 4 produse pentru a calcula cele două submatrice s şi t ai căror termeni principali sunt ag, bh, ce şi df. Să observăm că P_1 calculează termenul principal ag, P_2 calculează termenul principal bh, P_3 calculează termenul principal ce, iar P_4 calculează termenul principal df. Astfel, mai trebuie să se calculeze cele două submatrice rămase r şi u, ai căror termeni principali sunt termenii diagonali ae, bf, cg şi dh, fără a folosi mai mult de trei produse suplimentare. Încercăm să-l definim pe P_5 în vederea calculării a doi termeni principali deodată:

$$P_5 \;=\; A_5B_5 \;=\; (a+d)\cdot(e+h) \;=\; ae+ah+de+dh \;=\; \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}.$$

Pe lângă calculul celor doi termeni principali ae şi dh, P_5 calculează termenii auxiliari ah şi de, care trebuie să fie anihilaţi printr-o anumită modalitate. Putem să-i folosim pe P_4 şi P_2 pentru acest lucru, dar atunci apar alţi doi termeni auxiliari:

$$P_5+P_4-P_2 \;=\; ae+dh+df-bh \;=\; \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \end{pmatrix}.$$

Adăugând un produs suplimentar

$$P_6 \;=\; A_6B_6 \;=\; (b-d)\cdot(f+h) \;=\; bf+bh-df-dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix},$$

obţinem

$$r = P_5+P_4-P_2+P_6 = ae+bf = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

Putem să-l obţinem pe u, într-o manieră similară, din P_5, utilizând P_1 şi P_3 pentru a muta termenii auxiliari ai lui P_5 în altă direcţie:

$$P_5 + P_1 - P_3 = ae + ag - ce + dh = \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Scăzând un produs suplimentar

$$P_7 = A_7 B_7 = (a - c) \cdot (e + g) = ae + ag - ce - cg = \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

vom obţine

$$u = P_5 + P_1 - P_3 - P_7 = cg + dh = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Cele 7 produse de submatrice $P_1, P_2, \ldots, P_7$ pot fi utilizate pentru a calcula produsul $C = AB$, ceea ce completează descrierea metodei lui Strassen.

Discuţie

Timpul de execuţie mare al algoritmului lui Strassen îl face impracticabil dacă matricele nu sunt mari (n să fie cel puţin 45) şi nu sunt dense (puţine elemente nule). Pentru matrice mici, este preferabil algoritmul direct, iar pentru matrice mari, cu număr mare de elemente nule, există algoritmi speciali care sunt mai eficienţi decât algoritmul lui Strassen. În felul acesta, metoda lui Strassen este, în mare măsură, de interes teoretic.

Prin utilizarea tehnicilor avansate care depăşesc scopul acestei cărţi, se pot înmulţi matrice de ordinul $n \times n$ într-un timp mai bun decât $\Theta(n^{\lg 7})$. Cea mai bună limită superioară curentă este aproximativ egală cu $O(n^{2.376})$. Cea mai bună limită inferioară cunoscută este chiar limita evidentă $\Omega(n^2)$ (evident, deoarece trebuie să calculăm n^2 elemente ale matricei produs). În felul acesta, nu ştim cât de costisitoare este, în realitate, înmulţirea matricelor.

Algoritmul lui Strassen nu cere ca elementele matricelor să fie numere reale. Tot ce contează este faptul că sistemul de numere trebuie să formeze un inel algebric. Dacă elementele matricei nu formează un inel, pot fi introduse alte tehnici care să permită aplicarea metodei lui Strassen. Aceste rezultate se tratează în secţiunea următoare.

Exerciţii

31.2-1 Utilizaţi algoritmul lui Strassen pentru a calcula produsul matricelor

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}.$$

31.2-2 Cum s-ar modifica algoritmul lui Strassen pentru înmulţirea matricelor de ordinul $n \times n$ în care n nu este o putere a lui 2? Arătaţi că algoritmul rezultat se execută într-un timp de ordinul $\Theta(n^{\lg 7})$.

31.2-3 Care este cel mai mare k, astfel încât, dacă se pot înmulţi matrice de ordinul 3×3, folosind k înmulţiri (fără a presupune comutativitatea înmulţirii), să se poată înmulţi matrice de ordinul $n \times n$ într-un timp de ordinul $o(n^{\lg 7})$? Care ar fi timpul de execuţie al acestui algoritm?

31.2-4 V. Pan a descoperit un mod de a înmulţi matrice de ordinul 68×68 folosind 132464 înmulţiri, un mod de a înmulţi matrice de ordinul 70×70 folosind 143640 înmulţiri şi un mod de a înmulţi matrice de ordinul 72×72 folosind 155424 înmulţiri. Care metodă are cel mai bun timp asimptotic de execuţie când se utilizează un algoritm de înmulţire de matrice de tip divide şi stăpâneşte? Comparaţi acest timp cu timpul de execuţie pentru algoritmul lui Strassen.

31.2-5 Cât de repede se poate înmulţi o matrice de ordinul $kn \times n$ cu o matrice de ordinul $n \times kn$, utilizând algoritmul lui Strassen ca subrutină? Răspundeţi la aceeaşi întrebare inversând ordinea de intrare a matricelor.

31.2-6 Indicaţi modul de înmulţire a numerelor complexe $a + bi$ şi $c + di$ utilizând numai trei înmulţiri de numere reale. Algoritmul trebuie să-i aibă la intrare pe a, b, c şi d şi să producă, separat, componenta reală $ac - bd$ şi pe cea imaginară $ad + bc$.

31.3. Sisteme de numere algebrice şi înmulţirea matricelor booleene

Proprietăţile adunării şi înmulţirii matricelor depind de proprietăţile sistemului de numere peste care sunt definite matricele. În această secţiune definim trei sisteme diferite de numere care să fie elemente ale matricelor: cvasiinele, inele şi corpuri. Putem defini înmulţirea matricelor peste cvasiinele, dar algoritmul de înmulţire a matricelor al lui Strassen funcţionează peste inele. Prezentăm o tehnică simplă pentru a reduce înmulţirea matricelor booleene, care este definită peste un cvasiinel ce nu este un inel, la înmulţirea peste un inel. În final, vom arăta de ce proprietăţile unui corp nu pot fi exploatate în mod natural pentru a furniza algoritmi mai buni pentru înmulţirea matricelor.

Cvasiinele

Fie $(S, \oplus, \odot, \bar{0}, \bar{1})$ un sistem de numere, unde S este o mulţime de elemente, $\oplus$ şi $\odot$ sunt operaţii binare pe S (operaţii de adunare, respectiv, de înmulţire), iar $\bar{0}$ şi $\bar{1}$ sunt elemente distincte din S. Acest sistem este un ***cvasiinel*** dacă satisface următoarele proprietăţi:

1. $(S, \oplus, \bar{0})$ este un ***monoid***:
   - S este o mulţime ***închisă*** în raport cu $\oplus$; adică, $a \oplus b \in S$ pentru orice $a, b \in S$.
   - $\oplus$ este ***asociativă***, adică $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ pentru orice $a, b, c \in S$.
   - $\bar{0}$ este ***elementul neutru*** pentru $\oplus$, adică $a \oplus \bar{0} = \bar{0} \oplus a = a$ pentru orice $a \in S$.

La fel, $(S, \odot, \bar{1})$ este un monoid.

2. $\bar{0}$ este ***element neutru***, adică, $a \odot \bar{0} = \bar{0} \odot a = \bar{0}$ pentru orice $a \in S$.

3. Operatorul $\oplus$ este ***comutativ***, adică $a \oplus b = b \oplus a$ pentru orice $a, b \in S$.

4. Operatorul $\odot$ este ***distributiv*** în raport cu $\oplus$; adică $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ şi $(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$ pentru orice $a, b, c \in S$.

Exemple de cvasiinele sunt ***cvasiinelul boolean*** $(\{0,1\}, \vee, \wedge, 0, 1)$, unde $\vee$ notează operaţia SAU şi $\wedge$ notează operaţia logică ŞI, sistemul de numere naturale $(\mathbb{N}, +, \cdot, 0, 1)$, unde $+$ şi $\cdot$ notează operaţiile obişnuite de adunare şi înmulţire. Orice semiinel închis (vezi secţiunea 26.4) este, de asemenea, un cvasiinel; semi-inelele închise au, în plus, proprietăţile de idempotenţă şi sumă-infinită.

Putem extinde $\oplus$ şi $\odot$ pentru matrice aşa cum am făcut pentru $+$ şi $\cdot$ în secţiunea 31.1. Notăm matricea identitate de ordinul $n \times n$ compusă din $\bar{0}$ şi $\bar{1}$ prin $\bar{\mathrm{I}}_n$; găsim că înmulţirea matricelor este bine definită şi că sistemul de matrice este el însuşi un cvasiinel, aşa cum se afirmă în următoarea teoremă.

Teorema 31.7 (Matricele peste un cvasiinel formează un cvasiinel) Dacă $(S, \oplus, \odot, \bar{0}, \bar{1})$ este un cvasiinel şi $n \geq 1$, atunci $(S^{n \times n}, \oplus, \odot, \bar{0}, \bar{\mathrm{I}}_n)$ este un cvasiinel.

Demonstraţie. Demonstraţia este lăsată pe seama cititorului (exerciţiul 31.3-3). ∎

Inele

Scăderea nu este definită pentru cvasiinele, dar este definită pentru acele ***inele*** care sunt cvasiinele $(S, \oplus, \odot, \bar{0}, \bar{1})$ şi care satisfac următoarea proprietate suplimentară:

5. Orice element din S are un ***invers aditiv***, adică, pentru orice $a \in S$, există un element $b \in S$, astfel încât $a \oplus b = b \oplus a = \bar{0}$. Un astfel de b este numit ***negativul*** lui a şi este notat cu $(-a)$.

Fiind definit negativul unui element, putem defini scăderea prin $a - b = a + (-b)$.

Există multe exemple de inele. Mulţimea numerelor întregi $(\mathbb{Z}, +, \cdot, 0, 1)$, faţă de operaţiile obişnuite de adunare şi înmulţire, formează un inel. Mulţimea numerelor întregi modulo n pentru orice număr întreg $n > 1$ adică $(\mathbb{Z}_n, +, \cdot, 0, 1)$, unde $+$ este adunarea modulo n şi $\cdot$ este înmulţirea modulo n – formează un inel. Un alt exemplu este mulţimea $\mathbb{R}[x]$ a polinoamelor în x de grad finit având coeficienţi reali, faţă de operaţiile obişnuite – adică, $(\mathbb{R}[x], +, \cdot, 0, 1)$, unde $+$ este adunarea polinoamelor şi $\cdot$ este înmulţirea lor.

Corolarul următor arată că teorema 31.7 se generalizează în mod natural peste inele.

Corolarul 31.8 (Mulţimea matricelor peste un inel formează un inel) Dacă $(S, \oplus, \odot, \bar{0}, \bar{1})$ este un inel şi $n \geq 1$, atunci $(S^{n \times n}, \oplus, \odot, \bar{0}, \bar{\mathrm{I}}_n)$ este un inel.

Demonstraţie. Demonstraţia este lăsată pe seama cititorului (exerciţiul 31.3-3). ∎

Folosind acest corolar, putem demonstra următoarea teoremă.

Teorema 31.9 Algoritmul lui Strassen de înmulţire a matricelor funcţionează corect peste orice inel de matrice.

Demonstraţie. Algoritmul lui Strassen depinde de corectitudinea algoritmului pentru matrice de ordinul 2×2, algoritm care cere numai ca elementele matricelor să aparţină unui inel; corolarul 31.8 implică faptul că matricele însele formează un inel. Astfel, rezultă prin inducţie, că algoritmul lui Strassen funcţionează corect la fiecare nivel al recursivităţii. ∎

Algoritmul lui Strassen pentru înmulţirea matricelor, depinde, de fapt, esenţial de existenţa inversei aditive. Din cele şapte produse $P_1, P_2, \cdots, P_7$, patru implică diferenţă de submatrice. Astfel, în general, algoritmul lui Strassen nu va funcţiona pentru cvasiinele.

Înmulţirea matricelor booleene

Algoritmul lui Strassen nu se poate utiliza direct la înmulţirea matricelor booleene deoarece cvasiinelul boolean $(\{0,1\}, \vee, \wedge, 0, 1)$ nu este un inel. Există cazuri în care un cvasiinel este conţinut într-un sistem mai mare care este un inel. De exemplu, numerele naturale (un cvasiinel) sunt o submulţime a întregilor (un inel) şi de aceea, algoritmul lui Strassen poate fi utilizat pentru a înmulţi matrice de numere naturale în cazul în care considerăm că sistemul de numere peste care sunt definite matricele sunt întregii. Din păcate, cvasiinelul boolean nu poate fi extins într-un mod similar la un inel (vezi exerciţiul 31.3-4.)

Următoarea teoremă prezintă o metodă simplă pentru a reduce înmulţirea matricelor booleene la înmulţirea peste un inel. Problema 31-1 prezintă o astfel de concepţie eficientă.

Teorema 31.10 Dacă notăm cu $M(n)$ numărul operaţiilor aritmetice necesare pentru a înmulţi două matrice de ordinul $n \times n$ definite peste întregi, atunci două matrice booleene de ordinul $n \times n$ se pot înmulţi utilizând $O(M(n))$ operaţii aritmetice.

Demonstraţie. Fie A şi B cele două matrice, iar $C = AB$ aparţinând cvasiinelului boolean, adică,

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj}.$$

În loc de a calcula peste cvasiinelul boolean, vom calcula produsul C' peste inelul întregilor, folosind algoritmul de înmulţire a matricelor care utilizează $M(n)$ operaţii aritmetice. În felul acesta avem

$$c'_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Fiecare termen $a_{ik}b_{kj}$ al acestei sume este 0 dacă şi numai dacă $a_{ik} \wedge b_{kj} = 0$ şi 1 dacă şi numai dacă $a_{ik} \wedge b_{kj} = 1$. Astfel, suma întreagă c'_{ij} este 0 dacă şi numai dacă fiecare termen este 0 sau echivalent, dacă şi numai dacă SAU logic al termenilor care îl exprimă pe c_{ij} este 0. De aceea, matricea C poate fi reconstruită cu $\Theta(n^2)$ operaţii aritmetice din matricea de numere întregi C' prin simpla comparare cu 0 a fiecărui c'_{ij}. Numărul operaţiilor aritmetice, pentru întreaga procedură, este $O(M(n)) + \Theta(n^2) = O(M(n))$ deoarece $M(n) = \Omega(n^2)$. ∎

Astfel, folosind algoritmul lui Strassen, putem realiza înmulţirea matricelor booleene într-un timp de ordinul $O(n^{\lg 7})$.

Metoda normală de înmulţire a matricelor booleene utilizează numai variabile booleene. Dacă folosim această adaptare a algoritmului lui Strassen, matricea produs finală poate avea elemente

de mărimea lui n, deci, pentru a le memora, va fi nevoie de un cuvânt calculator, faţă de un singur bit. Mai îngrijorător este faptul că rezultalele intermediare, care sunt întregi, pot fi chiar mai mari. Pentru a nu avea rezultate intermediare care să crească prea mult trebuie realizat toate calculele modulo $n+1$. Exerciţiul 31.3-5 cere să se arate că, lucrând modulo $n+1$, nu este afectată corectitudinea algoritmului.

Corpuri

Un inel $(S, \oplus, \odot, \bar{0}, \bar{1})$ este un ***corp*** dacă sunt satisfăcute următoarele două proprietăţi suplimentare:

6. Operatorul $\odot$ este ***comutativ***, adică $a \odot b = b \odot a$ pentru orice $a, b \in S$.

7. Orice element nenul din S are un ***invers multiplicativ***; adică, pentru orice $a \in S - \{\bar{0}\}$, există un element $b \in S$ astfel ca $a \odot b = b \odot a = \bar{1}$. Un astfel de element b este adesea numit ***inversul*** lui a şi este notat cu a^{-1}.

Exemple de corpuri sunt numerele reale $(\mathbb{R}, +, \cdot, 0, 1)$, numerele complexe $(\mathbb{C}, +, \cdot, 0, 1)$ şi numerele întregi modulo un număr prim p: $(\mathbb{Z}_p, +, \cdot, 0, 1)$.

Deoarece într-un corp toate elementele au invers multiplicativ, împărţirea este posibilă. Operaţia este comutativă. Prin generalizare de la cvasiinele la inele, Strassen a reuşit să îmbunătăţească timpul de execuţie al algoritmului de înmulţire a două matrice. Întrucât elementele matricelor sunt adesea dintr-un corp, de exemplu numere reale, s-ar putea spera ca, utilizând corpuri în locul inelelor într-un algoritm recursiv de felul celui al lui Strassen, timpul de execuţie să fie în continuare îmbunătăţit.

Această concepţie pare puţin probabil de a fi rodnică. Pentru ca un algoritm recursiv divide şi stăpâneşte bazat pe corpuri să funcţioneze, matricele trebuie să formeze un corp la fiecare pas al recurenţei. Din nefericire, extinderea naturală a teoremei 31.7 şi a corolarului 31.8 la corpuri nu este posibilă. Pentru $n > 1$, mulţimea matricelor de ordinul $n \times n$ nu formează niciodată un corp, chiar dacă elementele acestora formează un corp. Înmulţirea matricelor de ordinul $n \times n$ nu este comutativă, iar multe matrice de ordinul $n \times n$ nu au inverse. De aceea, este mai probabil ca algoritmii mai buni pentru înmulţirea matricelor să se bazeze pe teoria inelelor şi nu pe cea a corpurilor.

Exerciţii

31.3-1 $\star$ Funcţionează oare algoritmul lui Strassen peste sistemul de numere $(\mathbb{Z}[x], +, \cdot, 0, 1)$, unde $\mathbb{Z}[x]$ este mulţimea tuturor polinoamelor cu coeficienţi întregi de variabilă x, iar $+$ şi $\cdot$ sunt operaţiile obişnuite de adunare şi înmulţire?

31.3-2 $\star$ Explicaţi de ce algoritmul lui Strassen nu funcţionează peste semi-inele închise (vezi secţiunea 26.4) sau peste semi-inelul boolean $(\{0, 1\}, \vee, \wedge, 0, 1)$.

31.3-3 $\star$ Demonstraţi teorema 31.7 şi corolarul 31.8.

31.3-4 $\star$ Arătaţi că cvasiinelul $(\{0, 1\}, \vee, \wedge, 0, 1)$ nu poate fi inclus într-un inel. Adică, arătaţi că este imposibil să se adauge "-1" la cvasiinel, astfel încât structura algebrică rezultată să fie un inel.

31.3-5 Argumentaţi că, dacă toate calculele din algoritmul teoremei 31.10 se realizează modulo $n+1$, algoritmul îmai funcţionează corect.

31.3-6 Arătaţi cum se determină eficient dacă un graf neorientat dat conţine un triunghi (o mulţime de trei vârfuri mutual adiacente).

31.3-7 ⋆ Arătaţi că produsul a două matrice booleene de ordinul $n \times n$ peste cvasiinelul boolean se reduce la calculul închiderii tranzitive a unui graf orientat dat, având $3n$ vârfuri.

31.3-8 Arătaţi cum se calculează închiderea tranzitivă a unui graf orientat dat, având n vârfuri, într-un timp de ordinul $O(n^{\lg 7} \lg n)$. Comparaţi acest rezultat cu performanţa procedurii ÎNCHIDERE-TRANZITIVĂ din secţiunea 26.2.

31.4. Rezolvarea sistemelor de ecuaţii liniare

Rezolvarea unui sistem de ecuaţii liniare este o problemă fundamentală care apare în diverse aplicaţii. Un sistem de ecuaţii liniare poate fi exprimat printr-o ecuaţie matriceală în care fiecare element al matricei sau vectorului aparţine unui corp, care, de obicei, este corpul $\mathbb{R}$ al numerelor reale. Această secţiune tratează modul de rezolvare a unui sistem de ecuaţii liniare utilizând o metodă numită desompunerea LUP.

Vom începem cu un sistem de ecuaţii liniare cu n necunoscute $x_1,\ x_2,\ \cdots,\ x_n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2, \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n. \end{aligned} \tag{31.17}$$

Despre un sistem de valori pentru $x_1,\ x_2,\ \ldots,\ x_n$ care satisfac, simultan, toate ecuaţiile (31.17) se spune că este o ***soluţie*** a acestui sistem de ecuaţii. În această secţiune vom trata numai cazul în care există exact n ecuaţii cu n necunoscute.

Putem să rescriem, în mod convenabil, ecuaţiile (31.17) ca o ecuaţie matrice-vector

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

sau, sub o formă echivalentă, notând cu $A = (a_{ij}), x = (x_j), b = (b_i)$, vom avea

$$Ax = b. \tag{31.18}$$

Dacă A este nesingulară, atunci matricea are o inversă A^{-1} şi

$$x = A^{-1}b \tag{31.19}$$

este vectorul soluţie. Putem demonstra că x este unica soluţie a ecuaţiei (31.18) după cum urmează. Dacă ar exista două soluţii, x şi x', atunci $Ax = Ax' = b$ şi

$$x = (A^{-1}A)x = A^{-1}(Ax) = A^{-1}(Ax') = (A^{-1}A)x' = x'.$$

În această secţiune, ne vom ocupa de cazul în care A este nesingulară, ceea ce este echivalent (conform teoremei 31.1) cu faptul că rangul lui A este egal cu numărul n al necunoscutelor. Cu toate acestea, există alte posibilităţi care merită o discuţie atentă. Dacă numărul ecuaţiilor este mai mic decât numărul n al necunoscutelor, sau, mai general, dacă rangul lui A este mai mic decât n, atunci sistemul este ***nedeterminat***. Un sistem nedeterminat are, de obicei, o infinitate de soluţii (vezi exerciţiul 31.4-9), deşi el poate să nu aibă nici o soluţie dacă ecuaţiile sunt incompatibile. Dacă numărul ecuaţiilor depăşeşte numărul necunoscutelor, sistemul este ***supradeterminat*** şi poate să nu aibă nici o soluţie. Găsirea unei soluţii aproximative, bune pentru un sistem de ecuaţii liniare supradeterminat este o problemă importantă care este tratată în secţiunea 31.6.

Să revenim la problema noastră de rezolvare a sistemului $Ax = b$ de n ecuaţii cu n necunoscute. O posibilitate este de a calcula pe A^{-1} şi, apoi, înmulţind ambii membri cu A^{-1}, obţinând $A^{-1}Ax = A^{-1}b$ sau $x = A^{-1}b$. Această metodă suferă, în practică, de ***instabilitate numerică***: erorile de rotunjire tind să se acumuleze când se utilizează numere reprezentate în virgulă flotantă în locul numerelor reale ideale. Din fericire, există o altă concepţie – descompunerea LUP – care are o stabilitate numerică şi care are marele avantaj de a fi mai rapidă de, aproximativ, trei ori.

Privire generală asupra descompunerii LUP

Ideea aflată în spatele descompunerii LUP este de a găsi trei matrice L, U şi P de ordinul $n \times n$ astfel încât

$$PA = LU, \tag{31.20}$$

unde

- L este o matrice unitate inferior triunghiulară,
- U este o matrice superior triunghiulară, iar
- P este o matrice de permutare.

Spunem că matricele L, U şi P care satisfac ecuaţia (31.20) formează o ***descompunere LUP*** a matricei A. Vom arăta că orice matrice nesingulară A are o astfel de descompunere.

Avantajul determinării unei descompuneri LUP pentru o matrice A este acela că sistemele pot fi rezolvate mai rapid când sunt triunghiulare, cum este cazul ambelor matrice L şi U. Având determinată o descompunere LUP pentru A, putem rezolva ecuaţia (31.18) $Ax = b$, rezolvând numai sisteme liniare triunghiulare, după cum urmează. Înmulţind ambele părţi ale lui $Ax = b$ cu P, se obţine ecuaţia echivalentă $PAx = Pb$, care, având în vedere exerciţiul 31.1-2, înseamnă să se permute ecuaţiile (31.17). Înlocuind în (31.20), obţinem

$$LUx = Pb.$$

Putem, acum, să rezolvăm această ecuaţie rezolvând două sisteme liniare triunghiulare. Fie $y = Ux$, unde x este vectorul soluţie căutat. Întâi rezolvăm sistemul inferior triunghiular

$$Ly = Pb \tag{31.21}$$

pentru a determina vectorul necunoscut y printr-o metodă numită "substituţie înainte". Cunoscându-l pe y, rezolvăm apoi sistemul superior triunghiular

$$Ux = y \tag{31.22}$$

cu necunoscuta x printr-o metodă numită "substituţie înapoi". Vectorul x este soluţia noastră pentru $Ax = b$, întrucât matricea de permutare P este inversabilă (exerciţiul 31.1-2):

$$Ax = P^{-1}LUx = P^{-1}Ly = P^{-1}Pb = b.$$

În pasul următor vom arăta cum funcţionează substituţiile înainte şi înapoi şi apoi vom ataca problema calculului însuşi al descompunerii LUP.

Substituţiile înainte şi înapoi

Substituţia înainte poate rezolva sistemul inferior triunghiular (31.21) într-un timp de ordinul $\Theta(n^2)$, dându-se L, P şi b. Este convenabil să reprezentăm permutarea P printr-un tablou $\pi[1..n]$. Pentru $i = 1, 2, \ldots, n$, elementul $\pi[i]$ indică faptul că $P_{i,\pi[i]} = 1$ şi $P_{ij} = 0$ pentru $j \neq \pi[i]$. Astfel, în linia i şi coloana j a lui PA este elementul $a_{\pi[i],j}$, iar al i-lea element al lui Pb este $b_{\pi[i]}$. Întrucât L este o matrice unitate inferior triunghiulară, ecuaţia (31.21) poate fi rescrisă astfel

$$\begin{aligned} y_1 &= b_{\pi[1]}, \\ l_{21}y_1 + y_2 &= b_{\pi[2]}, \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]}, \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]}. \end{aligned}$$

Este cât se poate de evident că îl putem găsi direct pe y_1, din prima ecuaţie $y_1 = b_{\pi[1]}$. Având soluţia pentru y_1, o substituim în a doua ecuaţie şi obţinem

$$y_2 = b_{\pi[2]} - l_{21}y_1.$$

Acum, îi substituim atât pe y_1 cât şi pe y_2 în ecuaţia a treia şi obţinem

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2).$$

În general, substituim pe $y_1, y_2, \cdots, y_{i-1}$ "în faţă" în ecuaţia a i-a, pentru a-l găsi pe y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j.$$

Substituţia înapoi este similară cu substituţia înainte. Dându-se U şi y, rezolvăm întâi a n-a ecuaţie şi procedăm la fel, revenind la prima ecuaţie. Ca şi substituţia înainte, acest proces se execută într-un timp de ordinul $\Theta(n^2)$. Întrucât U este o matrice superior triunghiulară, putem rescrie sistemul (31.22) sub forma

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1, \\ u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2, \\ &\vdots \\ u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2}, \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1}, \\ u_{n,n}x_n &= y_n. \end{aligned}$$

Astfel, putem determina succesiv pe x_n, x_{n-1}, $\ldots$, x_1 după cum urmează:

$$\begin{aligned} x_n &= y_n/u_{nn}, \\ x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1}, \\ x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2}, \\ &\vdots \end{aligned}$$

sau, în general,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / u_{ii}.$$

Dându-se P, L, U şi b, procedura LUP-Soluţie îl determină pe x combinând substituţiile înainte şi înapoi. Presupunem că dimensiunea n apare în atributul $linii[L]$ şi că matricea de permutare P se reprezintă prin tabloul π; pseudocodul va fi:

LUP-Soluţie(L, U, π, b)
1: $n \leftarrow linii[L]$
2: **pentru** $i \leftarrow 1, n$ **execută**
3: $\quad y_i \leftarrow b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j$
4: **pentru** $i \leftarrow n, 1, -1$ **execută**
5: $\quad x_i \leftarrow (y_i - \sum_{j=i+1}^{n} u_{ij}x_j)/u_{ii}$
6: **returnează** x

Procedura LUP-Soluţie îl determină pe y utilizând substituţia înainte în liniile 2–3, apoi îl determină pe x folosind substituţia înapoi în liniile 4–5. Întrucât există un ciclu implicit la însumări, în fiecare ciclu **pentru**, timpul de execuţie este de ordinul $\Theta(n^2)$.

Ca un exemplu al acestei metode, considerăm sistemul de ecuaţii liniare definit de

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 0.1 \\ 12.5 \\ 10.3 \end{pmatrix},$$

unde

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{pmatrix}, b = \begin{pmatrix} 0.1 \\ 12.5 \\ 10.3 \end{pmatrix},$$

pe care dorim să-l rezolvăm în raport cu necunoscuta x. Descompunerea LUP este

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{pmatrix}, U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{pmatrix}, P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

(Cititorul poate verifica faptul că $PA = LU$.) Folosind substituţia înainte, putem rezolva $Ly = Pb$ în raport cu y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 10.3 \\ 12.5 \\ 0.1 \end{pmatrix},$$

obţinând

$$y = \begin{pmatrix} 10.3 \\ 6.32 \\ -5.569 \end{pmatrix}$$

calculând întâi pe y_1, apoi, pe y_2 şi, în final, pe y_3. Folosind substituţia înapoi, rezolvăm $Ux = y$ în raport cu x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10.3 \\ 6.32 \\ -5,569 \end{pmatrix},$$

obţinând în acest fel răspunsul dorit

$$x = \begin{pmatrix} 0.5 \\ -0.2 \\ 3.0 \end{pmatrix}$$

calculând întâi pe x_3, apoi, pe x_2 şi, în final, pe x_1.

Calculul descompunerii LU

Până acum am arătat că, dacă o descompunere LUP poate fi determinată pentru o matrice nesingulară A, atunci substituţiile înainte şi înapoi pot fi folosite pentru a rezolva sistemul de ecuaţii liniare $Ax = b$. Rămâne să arătăm cum se poate găsi eficient o descompunere LUP pentru A. Începem cu cazul în care matricea A de ordinul $n \times n$ este nesingulară şi P este absentă (ceea ce este echivalent cu $P = I_n$). În acest caz, trebuie să găsim o factorizare $A = LU$. Cele două matrice L şi U formează o ***descompunere LU*** a lui A.

Procesul prin care realizăm descompunerea LU se numeşte ***eliminare gaussiană***. Începem prin a scădea multiplicări ale primei ecuaţii din celelalte ecuaţii, astfel încât să se elimine prima variabilă din acele ecuaţii. Apoi, scădem multiplicări ale ecuaţiei a doua din cea de a treia şi din celelalte ecuaţii care urmează, astfel încât acum prima şi cea de a doua variabilă să fie eliminate din ele. Continuăm acest proces până când sistemul obţinut are o formă superior triunghiulară – de fapt s-a obţinut matricea U. Matricea L este alcătuită din multiplicatorii de linii care au cauzat eliminarea variabilelor.

Algoritmul de implementare a acestei strategii este recursiv. Dorim să construim o descompunere LU pentru o matrice A de ordinul $n \times n$ nesingulară. Dacă $n = 1$, atunci problema este rezolvată deoarece putem alege $L = I_1$ şi $U = A$. Pentru $n > 1$, îl descompunem pe A în patru părţi:

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

unde v este un vector coloană de dimensiune $n-1$, w^{T} este un vector linie de dimensiune $n-1$, iar A' este o matrice de ordinul $(n-1) \times (n-1)$. Apoi, utilizând algebra matriceală (se verifică

ecuaţiile prin simpla lor înmulţire), îl putem factoriza pe A, astfel:

$$A = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}.$$

Zerourile din prima şi a doua matrice a factorizării sunt, respectiv, vector linie şi coloană de dimensiune $n-1$. Termenul vw^{T}/a_{11}, format, considerând produsul exterior a lui v şi w şi împărţind fiecare element al rezultatului cu a_{11}, este o matrice de ordinul $(n-1) \times (n-1)$ care este în conformitate cu dimensiunea matricei A' din care este scăzut. Matricea rezultat:

$$A' - vw^{\mathrm{T}}/a_{11} \tag{31.23}$$

de ordinul $(n-1) \times (n-1)$ se numeşte ***complementul Schur*** al lui A în raport cu a_{11}.

Vom determina acum recursiv o descompunere LU a complementului Schur. Fie

$$A' - vw^{\mathrm{T}}/a_{11} = L'U',$$

unde L' este matricea unitate inferior triunghiulară şi U' este matrice superior triunghiulară. Atunci, utilizând algebra matricelor, avem

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} = LU, \end{aligned}$$

ceea ce demonstrează existenţa descompunerii LU. (Să observăm că, deoarece L' este matrice unitate inferior triunghiulară, şi L este o matrice inferior triunghiulară, şi că, deoarece U' este matrice superior triunghiulară, la fel este şi U.)

Desigur, dacă $a_{11} = 0$, această metodă nu funcţionează, din cauza împărţirii cu zero. De asemenea, ea nu funcţionează dacă elementul superior cel mai din stânga al complementului Shur $A' - vw^{\mathrm{T}}/a_{11}$ este zero, deoarece împărţim cu el în pasul următor al recurenţei. Elementele cu care împărţim în timpul descompunerii LU se numesc elemente ***pivot*** şi ocupă diagonala matricei U. Raţiunea de a include matricea de permutare P, în timpul descompunerii LUP, este aceea că ea ne permite să eliminăm împărţirea cu zero. Utilizarea permutărilor, pentru a evita împărţirea cu zero (sau cu valori mici), se numeşte ***pivotare***.

O clasă importantă de matrice pentru care descompunerea LU funcţionează întotdeauna corect este clasa matricelor simetrice pozitiv-definite. Astfel de matrice nu necesită pivotare şi, în felul acesta, strategia recursivă schiţată mai sus poate fi utilizată fără frica împărţirii cu 0. Vom demonstra acest rezultat, împreună cu altele, în secţiunea 31.6.

Codul pentru descompunerea LU a unei matrice A urmează strategia recursivă, exceptând cazul când un ciclu iterativ înlocuieşte recurenţa. (Această transformare este o optimizare standard pentru o procedură "recursivă-la-sfârşit" – una a cărei ultimă operaţie este un apel recursiv la ea însăşi.) Se presupune că dimensiunea lui A este păstrată în atributul $linii[A]$. Întrucât ştim că matricea de ieşire U are zerouri sub diagonală şi că LU-SOLUŢIE nu utilizează aceste intrări, codul nu se ocupă de completarea lor. La fel, deoarece matricea de ieşire L are 1 pe diagonală şi 0 deasupra ei, aceste elemente nu sunt completate. Astfel, codul sursă calculează numai elementele "semnificative" ale lui L şi U.

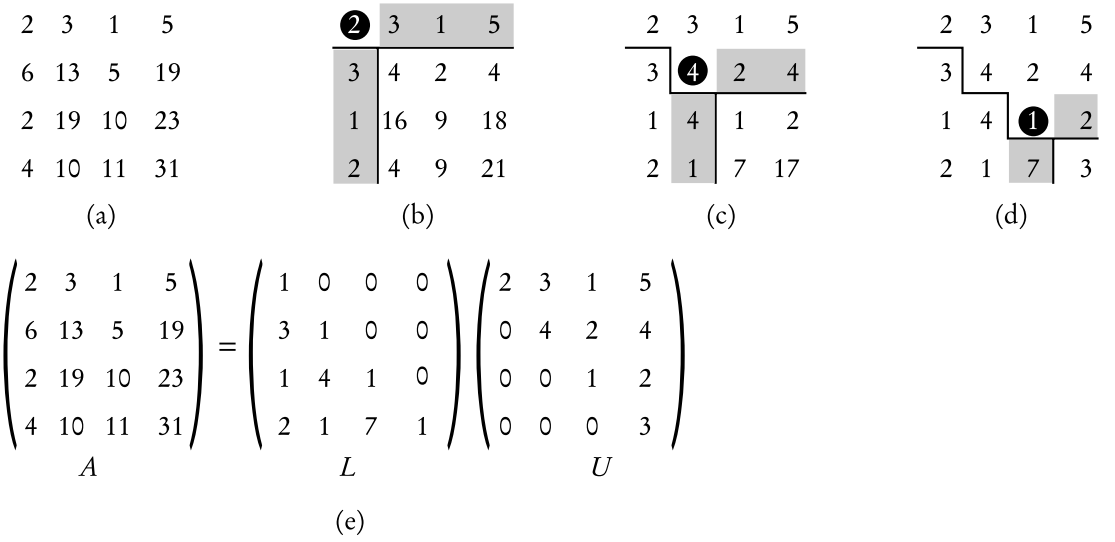


Figura 31.1 Operaţia LU-Descompunere. (**a**) Matricea A. (**b**) Elementul $a_{11} = 2$ haşurat cu negru este pivotul, coloana haşurată cu gri este v/a_{11}, iar linia haşurată cu gri este w^{T}. Elementele lui U calculate astfel sunt situate deasupra liniei orizontale, iar elementele lui L sunt în stânga liniei verticale. Matricea complementul Schur $A' - vw^{\mathrm{T}}/a_{11}$ se află în dreapta jos. (**c**) Acum operăm asupra matricei complement Schur produsă în partea (b). Elementul $a_{22} = 4$ haşurat cu negru este pivotul, iar coloana şi linia haşurate cu gri sunt respectiv v/a_{22} şi w^{T} (în partiţionarea complementului Schur). Liniile împart matricea în elementele lui U calculate până acum (deasupra), elementele lui L calculate până acum (în stânga) şi complementul Schur nou (dreapta jos). (**d**) Pasul următor completează factorizarea (Elementul 3 din complementul Schur nou devine parte a lui U când se termină recurenţa.) (**e**) Factorizarea $A = LU$.

LU-Descompunere(A)

```
 1: n ← linii[A]
 2: pentru k ← 1, n execută
 3:     u_kk ← a_kk
 4:     pentru i ← k + 1, n execută
 5:         l_ik ← a_ik/u_kk            ▷ l_ik păstrează v_i
 6:         u_ki ← a_ki                 ▷ u_ki păstrează w_i^T
 7:     pentru i ← k + 1, n execută
 8:         pentru j ← k + 1, n execută
 9:             a_ij ← a_ij − l_ik u_kj
10: returnează L şi U
```

Ciclul exterior **pentru**, care începe în linia a doua, iterează o dată pentru fiecare pas recursiv. În acest ciclu, pivotul se determină în linia 3 şi este $u_{kk} = a_{kk}$. În ciclul **pentru** din liniile 4–6 (care nu se execută când $k = n$), vectorii v şi w^{T} se utilizează pentru a-i actualiza pe L şi U. Elementele vectorului v se determină în linia 5, unde v_i este memorat în l_{ik}, iar elementele vectorului w^{T} se determină în linia 6, unde w_i^{T} se memorează în u_{ki}. În final, elementele complementului Schur se calculează în liniile 7–9 şi se păstrează tot în matricea A. Deoarece linia 9 este triplu imbricată, LU-Descompunere se execută într-un timp de ordinul $\Theta(n^3)$.

Figura 31.1 ilustrează operaţia LU-DESCOMPUNERE. Ea arată o optimizare standard a procedurii în care elementele semnificative ale lui L şi U se păstrează "în poziţii" din matricea A. Adică, putem stabili o corespondenţă între fiecare element a_{ij} şi l_{ij} (dacă $i > j$) sau u_{ij} (dacă $i \leq j$) şi să actualizăm matricea A, astfel încât ea să păstreze atât pe L cât şi pe U, când se termină procedura. Pseudocodul pentru această optimizare se obţine simplu din pseudocodul de mai sus, înlocuind fiecare referinţă la l sau u prin a; nu este dificil să se verifice că această transformare este corectă.

Calculul unei descompuneri LUP

În general, pentru rezolvarea unui sistem de ecuaţii liniare $Ax = b$, trebuie să pivotăm elemente din afara diagonalei lui A pentru a elimina împărţirea cu 0. Nu doar împărţirea cu 0 este indezirabilă, ci şi împărţirea cu valori mici, chiar dacă A este nesingulară, deoarece instabilităţile numerice pot rezulta din calcule. De aceea, încercăm să pivotăm o valoare mare.

Matematica aflată în spatele descompunerii LUP este similară celei din esenţa descompunerii LU. Amintim că se dă o matrice nesingulară A de ordinul $n \times n$ şi dorim să găsim o matrice de permutare P, o matrice unitate inferior triunghiulară L şi o matrice superior triunghiulară U, astfel încât $PA = LU$. Înainte de a partiţiona matricea A, aşa cum am făcut pentru descompunerea LU, vom muta un element nenul, de exemplu a_{k1}, din prima coloană în poziţia (1,1) a matricei. (Dacă prima coloană conţine numai zerouri, atunci A este singulară, deoarece determinantul ei este 0, prin teoremele 31.4 şi 31.5.) În vederea conservării sistemului de ecuaţii, schimbăm linia 1 cu linia k, ceea ce este echivalent cu înmulţirea la stânga a lui A cu o matrice de permutare Q (exerciţiul 31.1-2). Astfel, pe QA îl putem scrie:

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

unde $v = (a_{21}, a_{31}, \ldots, a_{n1})^{\mathrm{T}}$, exceptând pe a_{11} care îl înlocuieşte pe a_{k1}; $w^{\mathrm{T}} = (a_{k2}, a_{k3}, \ldots, a_{kn})$; iar A' este o matrice de ordinul $(n-1) \times (n-1)$. Deoarece $a_{k1} \neq 0$, putem realiza, în mare parte, aceeaşi algebră liniară ca şi pentru descompunerea LU, dar având garanţia că nu vom împărţi prin 0:

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix}.$$

Complementul Schur $A' - vw^{\mathrm{T}}/a_{k1}$ este nesingular, deoarece, dacă matricea a doua din ultima ecuaţie ar avea determinantul 0, atunci şi determinantul matricei A ar fi 0; dar aceasta înseamnă că A este singulară. În consecinţă, putem găsi inductiv o descompunere LUP pentru complementul Schur, cu o matrice unitate inferior triunghiulară L', o matrice superior triunghiulară U' şi o matrice permutare P', astfel încât

$$P'(A' - vw^{\mathrm{T}}/a_{k1}) = L'U'.$$

Definim pe

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q,$$

care este o matrice de permutare pentru că este produsul a două matrice de permutare (exerciţiul 31.1-2). Avem

$$\begin{aligned}
PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\
&= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & P'(A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} = LU,
\end{aligned}$$

adică descompunrea LUP. Deoarece L' şi L sunt matrice inferior triunghiulare şi din cauză că U' este superior triunghiulară, la fel este şi U.

Să observăm că, în această derivare, spre deosebire de cea pentru descompunerea LU, atât vectorul coloană v/a_{k1} cât şi complementul Schur $A' - vw^{\mathrm{T}}/a_{k1}$ trebuie să fie înmulţiţi cu matricea de permutare P'.

LUP-Descompunere(A)

```
1:  n ← linii[A]
2:  pentru i ← 1, n execută
3:     π[i] ← i
4:  pentru k ← 1, n execută
5:     p ← 0
6:     pentru i ← k, n execută
7:        dacă |a_ik| > p atunci
8:           p ← |a_ik|
9:           k' ← i
10:       dacă p = 0 atunci
11:          eroare "matrice singulară"
12:       interschimbă π[k] ↔ π[k']
13:       pentru i ← 1, n execută
14:          interschimbă a_ki ↔ a_k'i
15:       pentru i ← k + 1, n execută
16:          a_ik ← a_ik/a_kk
17:          pentru j ← k + 1, n execută
18:             a_ij ← a_ij − a_ik a_kj
```

Ca şi în cazul algoritmului LU-Descompunere, pseudocodul pentru descompunerea LUP înlocuieşte recurenţa cu un ciclu iterativ. Ca o îmbunătăţire faţă de implementarea direct recursivă, păstrăm dinamic matricea de permutare P printr-un tablou π, unde $\pi[i] = j$ înseamnă că linia a i-a a lui P conţine 1 în coloana j. De asemenea, implementăm codul pentru calculul

lui L şi U "în spaţiul" matricei A. În felul acesta, când procedura se termină,

$$a_{ij} = \begin{cases} l_{ij} & \text{dacă } i > j, \\ u_{ij} & \text{dacă } i \le j. \end{cases}$$

Figura 31.2 ilustrează modul în care LUP-DESCOMPUNERE factorizează o matrice. Vectorul π se iniţializează în liniile 2–3 pentru a reprezenta permutarea identică. Ciclul **pentru** exterior care începe în linia 4 implementează recurenţa. La fiecare iteraţie a ciclului exterior, liniile 5–9 determină elementul $a_{k'k}$ având cea mai mare valoare absolută dintre elementele aflate în prima coloană curentă (coloana k) din matricea de ordin $(n-k+1) \times (n-k+1)$ a cărei descompunere LU trebuie găsită.

Dacă toate elementele din prima coloană curentă sunt zero, liniile 10–11 stabilesc că matricea este singulară. Pentru pivotare, în linia 12 se interschimbă $\pi[k']$ cu $\pi[k]$, şi în liniile 13–14 se interschimbă al k-lea şi al k'-lea rând al lui A, în felul acesta a_{kk} devine elementul pivot. (Rândurile întregi se permută deoarece, în derivarea de mai sus, nu doar $A' - vw^{\mathrm{T}}/a_{k1}$ se înmulţeşte cu P', ci şi v/a_{k1}.) În final, complementul Schur este calculat prin liniile 15–18 în mare parte în acelaşi mod în care a fost calculat prin liniile 4–9 ale algoritmului LUP-DESCOMPUNERE, exceptând faptul că aici operaţia este scrisă pentru a funcţiona în "spaţiul" matricei A.

Din cauza structurii ciclului triplu imbricat, timpul de execuţie al algoritmului LUP-DESCOMPUNERE este $\Theta(n^3)$, acelaşi ca şi pentru LU-DESCOMPUNERE. Astfel, pivotarea ne costă, în timp, cel mult un factor constant.

Exerciţii

31.4-1 Rezolvaţi ecuaţia

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

folosind substituţia înainte.

31.4-2 Găsiţi o descompunere LU a matricei

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

31.4-3 De ce ciclul **pentru** din linia 4 a algoritmului LUP-DESCOMPUNERE se execută numai până la $n-1$, în timp ce ciclul **pentru**, corespunzător, din linia 2 a algoritmului LU-DESCOMPUNERE se execută în acelaşi mod până la n?

31.4-4 Rezolvaţi ecuaţia

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

folosind o descompunere LUP.

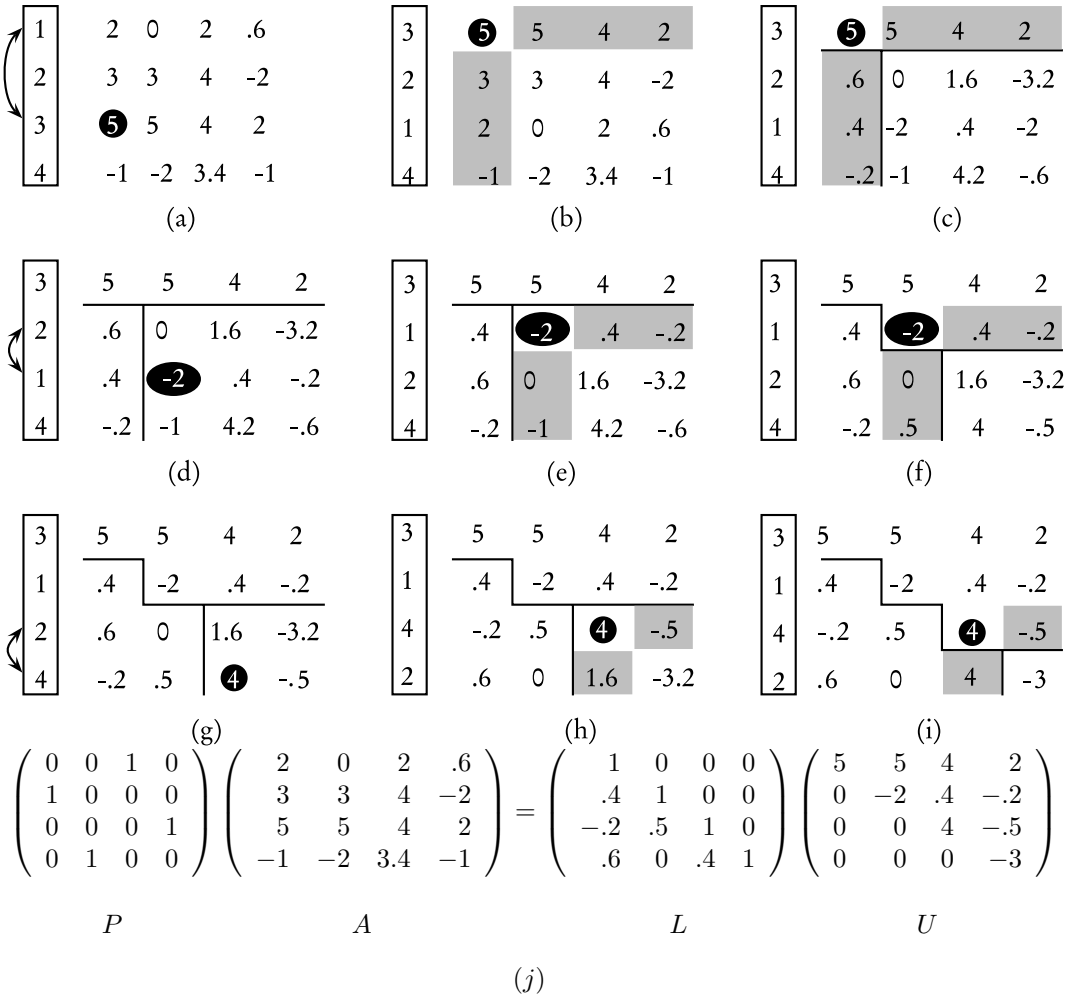


Figura 31.2 Operaţia de LUP-Descompunere. (**a**) Matricea de intrare A cu permutarea identică a liniilor în stânga. Primul pas al algoritmului determină faptul că elementul 5, din linia a treia, haşurat cu negru, este pivotul pentru prima coloană. (**b**) Rândurile 1 şi 3 se permută şi permutarea este actualizată. Coloana şi linia haşurate cu gri reprezintă pe v şi w^{T}. (**c**) Vectorul v este înlocuit cu $v/5$ şi partea din dreapta mai jos a matricei este actualizată cu complementul Schur. Liniile împart matricea în trei regiuni: elementele lui U (deasupra), elementele lui L (în stânga) şi elementele complementului lui Schur (în dreapta jos). (**d**)-(**f**) Pasul al doilea. (**g**)-(**i**) Pasul al treilea. Nu mai intervin modificări în pasul patru. (**j**) Descompunerea LUP $PA = LU$.

31.4-5 Descrieţi descompunerea LUP a unei matrice diagonale.

31.4-6 Descrieţi descompunerea LUP a unei matrice de permutare A şi demonstraţi că ea este unică.

31.4-7 Arătaţi că, pentru orice $n \geq 1$, există matrice singulare de ordinul $n \times n$ care au descompuneri LU.

31.4-8 $\star$ Arătaţi cum se poate rezolva eficient un sistem de ecuaţii de forma $Ax = b$ peste un cvasiinel boolean $(\{0,1\}, \vee, \wedge, 0, 1)$.

31.4-9 $\star$ Să presupunem că A este o matrice reală de ordinul $m \times n$ de rang m, unde $m < n$. Arătaţi cum se poate determina un vector n-dimensional x_0 şi o matrice de ordinul $m \times (n-m)$ şi de rang $n-m$, astfel încât orice vector de forma $x_0 + By$, pentru $y \in \mathbb{R}^{n-m}$, este o soluţie a ecuaţiei nedeterminate $Ax = b$.

31.5. Inversarea matricelor

Deşi în practică, în general, nu vom folosi inversele matricelor pentru a rezolva sisteme de ecuaţii liniare, preferând folosirea de tehnici numerice mai stabile de felul descompunerii LUP, uneori fiind necesar să se calculeze inversa unei matrice. În această secţiune, vom arăta cum poate fi folosită descompunerea LUP pentru a calcula inversa unei matrice. De asemenea, vom discuta problema teoretică interesantă, dacă se poate sau nu, accelera calculul inversei unei matrice, utilizând tehnici de felul algoritmului lui Strassen pentru înmulţirea matricelor. Într-adevăr, articolul originar al lui Strassen avea ca motivaţie de a arăta că un sistem de ecuaţii liniare ar putea fi rezolvat mai rapid decât prin metoda obişnuită.

Calculul inversei unei matrice dintr-o descompunere LUP

Să presupunem că avem o descompunere LUP a unei matrice A sub forma a trei matrice L, U şi P, astfel încât $PA = LU$. Folosind algoritmul LU-Soluţie, putem rezolva o ecuaţie de forma $Ax = b$ într-un timp de ordinul $\Theta(n^2)$. În general, o dată ce avem descompunerea LUP a lui A, putem rezolva, într-un timp de ordinul $\Theta(kn^2)$ k versiuni de ecuaţii $Ax = b$ care diferă numai prin b.

Ecuaţia

$$AX = I_n \tag{31.24}$$

poate fi privită ca un set de n ecuaţii distincte de forma $Ax = b$. Aceste ecuaţii definesc matricea X ca inversă a lui A. Pentru a fi mai exacţi, notăm cu X_i a i-a coloană a lui X şi amintim că vectorul unitate e_i este a i-a coloană a lui I_n. Ecuaţia (31.24) poate fi, apoi, rezolvată în X folosind descompunerea LUP a lui A pentru a rezolva, separat fiecare ecuaţie

$$AX_i = e_i$$

separat, pentru X_i. Fiecare dintre cei n de X_i poate fi găsit într-un timp de ordinul $\Theta(n^2)$ şi astfel calculul lui X prin descompunerea LUP a lui A ia un timp de ordinul lui $\Theta(n^3)$. Întrucât descompunerea LUP a lui A poate fi determinată într-un timp de ordinul lui $\Theta(n^3)$, inversa A^{-1} a matricei A poate fi determinată într-un timp de ordinul lui $\Theta(n^3)$.

Înmulţirea şi inversarea matricelor

Arătăm acum că accelerarea teoretică obţinută pentru înmulţirea matricelor se traduce printr-o accelerare pentru inversarea matricelor. De fapt, demonstrăm ceva mai mult: inversarea matricelor este echivalentă cu înmulţirea, în următorul sens. Dacă notăm cu $M(n)$ timpul de înmulţire a două matrice de ordinul $n \times n$ şi cu $I(n)$ timpul de inversare a unei matrice nesingulare de acelaşi ordin, atunci $I(n) = \Theta(M(n))$. Demonstrăm acest rezultat în două etape. Întâi, arătăm că $M(n) = O(I(n))$, ceea ce este relativ simplu, iar apoi demonstrăm că $I(n) = O(M(n))$.

Teorema 31.11 (Înmulţirea nu este mai grea decât inversarea) Dacă putem inversa o matrice de ordinul $n \times n$ în timpul $I(n)$, unde $I(n) = \Omega(n^2)$ satisface condiţia de regularitate $I(3n) = O(I(n))$, atunci putem înmulţi două matrice de ordinul $n \times n$ într-un timp de ordinul $O(I(n))$.

Demonstraţie. Fie A şi B două matrice de ordinul $n \times n$. Vrem să calculăm matricea lor produs C. Definim matricea D de ordinul $3n \times 3n$ astfel

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

Inversa lui D este

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

şi, astfel, putem calcula produsul AB luând submatricea de ordinul $n \times n$ din dreapta sus a matricei D^{-1}.

Putem determina matricea D într-un timp de ordinul $\Theta(n^2) = O(I(n))$ şi o putem inversa într-un timp de ordinul $O(I(3n)) = O(I(n))$, prin condiţia de regularitate pentru $I(n)$. Astfel avem

$$M(n) = O(I(n)).$$

■

Să observăm că $I(n)$ satisface condiţia de regularitate doar dacă $I(n)$ nu are salturi mari în valoare. De exemplu, dacă $I(n) = \Theta(n^c \lg^d n)$, pentru orice constante $c > 0, d \geq 0$, atunci $I(n)$ satisface condiţia de regularitate.

Reducerea inversării matricelor la înmulţirea lor

Demonstraţia că inversarea matricelor nu este mai grea decât înmulţirea lor, se bazează pe câteva proprietăţi ale matricelor simetrice pozitiv-definite care vor fi studiate în secţiunea 31.6.

Teorema 31.12 (Inversarea nu este mai grea decât înmulţirea) Presupunând că putem înmulţi două matrice reale de ordinul $n \times n$ în timpul $M(n)$, unde $M(n) = \Omega(n^2)$ şi $M(n)$ satisface cele două condiţii de regularitate $M(n + k) = O(M(n))$ pentru orice k, $0 \leq k \leq n$, şi $M(n/2) \leq cM(n)$, pentru orice constantă $c < 1/2$, atunci putem calcula inversa oricărei matrice reale nesingulare de ordinul $n \times n$ în timpul $O(M(n))$.

Demonstraţie. Putem presupune că n este multiplu de 2, deoarece

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix},$$

pentru orice $k > 0$. Aşadar, alegând pe k astfel încât $n + k$ să fie o putere a lui 2, lărgim matricea la o dimensiune care este puterea următoare a lui 2 şi obţinem răspunsul dorit, A^{-1}, din răspunsul la problema lărgită. Condiţia de regularitate asupra lui $M(n)$ asigură ca această lărgire să nu cauzeze creşterea timpului de execuţie mai mult decât cu un factor constant.

Pentru moment, să presupunem că matricea A de ordinul $n \times n$ este simetrică şi pozitiv-definită. O partiţionăm în patru submatrice de ordinul $n/2 \times n/2$:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} \tag{31.25}$$

Apoi, dacă

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{31.26}$$

este complementul Schur al lui A în raport cu B, atunci

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix} \tag{31.27}$$

deoarece $AA^{-1} = I_n$, ceea ce se poate verifica înmulţind matricele. Matricele B^{-1} şi S^{-1} există dacă A este simetrică şi pozitiv-definită, conform lemelor 31.13 şi 31.15 din secţiunea 31.6, deoarece atât B cât şi S sunt simetrice şi pozitiv-definite. Conform exerciţiului 31.1-3 avem că $B^{-1}C^{\mathrm{T}} = (CB^{-1})^{\mathrm{T}}$ şi $B^{-1}C^{\mathrm{T}}S^{-1} = (S^{-1}CB^{-1})^{\mathrm{T}}$. De aceea, ecuaţiile (31.26) şi (31.27) pot fi folosite pentru a specifica un algoritm recursiv care implică 4 înmulţiri de matrice de ordinul $n/2 \times n/2$:

$$\begin{aligned} &C \cdot B^{-1}, \\ &(C \cdot B^{-1}) \cdot C^{\mathrm{T}}, \\ &S^{-1} \cdot (CB^{-1}), \\ &(CB^{-1})^{\mathrm{T}} \cdot (S^{-1}CB^{-1}). \end{aligned}$$

Întrucât matricele de ordinul $n/2 \times n/2$ se înmulţesc folosind un algoritm pentru matrice de ordinul $n \times n$, inversarea matricelor simetrice pozitiv-definite se poate rezolva în timpul

$$I(n) \;\le\; 2I(n/2) + 4M(n) + O(n^2) \;=\; 2I(n/2) + O(M(n)) \;=\; O(M(n)).$$

Rămâne de demonstrat că timpul asimptotic de execuţie a înmulţirii matricelor poate fi obţinut pentru inversarea matricelor, dacă A este inversabilă, dar nu este simetrică şi pozitiv-definită. Ideea de bază este aceea că, pentru orice matrice A nesingulară, matricea $A^{\mathrm{T}}A$ este simetrică (vezi exerciţiul 31.1-3) şi pozitiv-definită (vezi teorema 31.6). Astfel, problema inversării lui A se reduce la problema inversării lui $A^{\mathrm{T}}A$.

Reducerea se bazează pe observaţia că, atunci când A este o matrice nesingulară de ordinul $n \times n$, avem

$$A^{-1} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}},$$

deoarece $((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})A = (A^{\mathrm{T}}A)^{-1}(A^{\mathrm{T}}A) = I_n$, şi inversa unei matrice este unică. Prin urmare, putem calcula A^{-1}, înmulţind întâi pe A^{T} cu A pentru a-l obţine pe $A^{\mathrm{T}}A$, inversând apoi matricea simetrică şi pozitiv-definită $A^{\mathrm{T}}A$ prin algoritmul divide şi stăpâneşte prezentat, şi, în final, înmulţind rezultatul cu A^{T}. Fiecare dintre aceşti paşi necesită un timp de ordinul $O(M(n))$. Astfel orice matrice nesingulară cu elemente reale poate fi inversată într-un timp de ordinul $O(M(n))$. ∎

Demonstraţia teoremei 31.12 sugerează un mod de a rezolva ecuaţii $Ax = b$ fără pivotare, atât timp cât A este nesingulară. Înmulţind ambii membri ai ecuaţiei cu A^{T}, obţinem $(A^{\mathrm{T}}A)x = A^{\mathrm{T}}b$. Această transformare nu afectează soluţia x deoarece A^{T} este inversabilă, aşa că putem factoriza matricea $A^{\mathrm{T}}A$ simetrică şi pozitiv-definită cu ajutorul descompunerii LU. Apoi, folosim substituţiile înainte şi înapoi pentru a o rezolva, în raport cu x, partea dreaptă fiind $A^{\mathrm{T}}b$. Deşi această metodă este teoretic corectă, în practică, procedura LUP-DESCOMPUNERE funcţionează mult mai bine. Descompunerea LUP necesită mai puţine operaţii aritmetice printr-un factor constant şi are proprietăţi numerice puţin mai bune.

Exerciţii

31.5-1 Fie $M(n)$ timpul de înmulţire a matricelor de ordinul $n \times n$, iar $S(n)$ timpul necesar pentru a ridica la pătrat o matrice de ordinul $n \times n$. Arătaţi că înmulţirea şi ridicarea la pătrat a matricelor au în esenţă aceeaşi dificultate: $S(n) = \Theta(M(n))$.

31.5-2 Fie $M(n)$ timpul de înmulţire a matricelor de ordinul $n \times n$ şi $L(n)$ timpul de calcul al descompunerii LUP pentru o matrice de ordinul $n \times n$. Arătaţi că înmulţirea matricelor şi calculul descompunerii LUP pentru o matrice au, în esenţă, aceeaşi dificultate: $L(n) = \Theta(M(n))$.

31.5-3 Fie $M(n)$ timpul de înmulţire a matricelor de ordinul $n \times n$, iar $D(n)$ timpul necesar pentru a calcula determinantul unei matrice de ordinul $n \times n$. Arătaţi că aflarea determinantului nu este mai grea decât înmulţirea matricelor: $D(n) = O(M(n))$.

31.5-4 Fie $M(n)$ timpul de înmulţire a matricelor booleene de ordinul $n \times n$, şi $T(n)$ timpul de determinare a închiderii tranzitive a matricelor booleene de ordinul $n \times n$. Arătaţi că $M(n) = O(T(n))$ şi $T(n) = O(M(n) \lg n)$.

31.5-5 Stabiliţi dacă funcţionează algoritmul de inversare a matricelor, bazat pe teorema 31.12, când elementele matricelor aparţin corpului de întregi modulo 2? Explicaţi.

31.5-6 ⋆ Generalizaţi algoritmul de inversare a matricelor din teorema 31.12 pentru a trata matrice de numere complexe şi demonstraţi că generalizarea respectivă funcţionează corect. (*Indicaţie:* în locul transpusei lui A, se utilizează ***transpusa conjugată*** A^*, care se obţine din transpusa lui A prin înlocuirea fiecărui element cu complex-conjugatul lui. În locul matricelor simetrice se consideră matrice ***hermitiene***, care sunt matrice A astfel încât $A = A^*$.)

31.6. Matrice simetrice pozitiv-definite şi aproximarea prin metoda celor mai mici pătrate

Matricele simetrice pozitiv-definite au multe proprietăţi interesante şi avantajoase. De exemplu, sunt nesingulare, iar descompunerea LU poate fi făcută fără să ne intereseze împărţirea cu zero. În această secţiune, vom demonstra diferite alte proprietăţi importante ale lor şi vom prezenta o aplicaţie importantă de ajustare a curbelor şi anume aproximarea prin metoda celor mai mici pătrate.

Prima proprietate, pe care o demonstrăm, este poate cea mai importantă.

Lema 31.13 Orice matrice simetrică pozitiv-definită este nesingulară.

Demonstraţie. Presupunem că A este o matrice singulară. Atunci, pe baza corolarului 31.3, rezultă că există un vector nenul x, astfel încât $Ax = 0$. Înseamnă că $x^{\mathrm{T}}Ax = 0$, deci, A nu poate fi pozitiv-definită. ∎

Demonstraţia faptului că putem realiza o descompunere LU a unei matrice A simetrice pozitiv-definite fără a împărţi la 0, este mai complicată. Începem prin a demonstra proprietăţi pentru anumite submatrice ale lui A. Definim matricea A_k, ***submatrice directoare*** a lui A, ca fiind formată din A din intersecţiile dintre primele k linii şi primele k coloane ale lui A.

Lema 31.14 Dacă A este o matrice simetrică pozitiv-definită, atunci fiecare submatrice directoare a lui A este simetrică şi pozitiv-definită.

Demonstraţie. Faptul că fiecare submatrice directoare A_k este simetrică, este evident. Pentru a demonstra că A_k este pozitiv-definită, fie x un vector coloană nenul de dimensiune k şi fie A partiţionată după cum urmează:

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}.$$

Atunci, avem

$$x^{\mathrm{T}}A_k x \;=\; (x^{\mathrm{T}} \quad 0)\begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}\begin{pmatrix} x \\ 0 \end{pmatrix} \;=\; (x^{\mathrm{T}} \quad 0)A\begin{pmatrix} x \\ 0 \end{pmatrix} \;>\; 0$$

deoarece A este pozitiv-definită, rezultă că şi A_k este pozitiv-definită. ∎

Ne întoarcem la câteva proprietăţi esenţiale ale complementului Schur. Fie A o matrice simetrică pozitiv-definită şi A_k submatricea directoare de ordinul $k \times k$ a lui A. A este partiţionată astfel

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}. \tag{31.28}$$

Complementul Schur al lui A, în raport cu A_k, se defineşte prin

$$S = C - BA_k^{-1}B^{\mathrm{T}}. \tag{31.29}$$

(Conform lemei 31.14, A_k este simetrică şi pozitiv-definită, astfel, pe baza lemei 31.13 există A_k^{-1}, rezultă că S este bine definit.) Să observăm că definiţia (31.23) a complementului Schur este compatibilă cu definiţia (31.29) pentru $k = 1$.

Lema următoare arată că matricele complement Schur ale matricelor simetrice pozitiv-definite sunt simetrice şi pozitiv-definite. Acest rezultat a fost utilizat în teorema 31.12, iar corolarul ei a fost necesar pentru a demonstra corectitudinea descompunerii LU pentru matrice simetrice pozitiv-definite.

Lema 31.15 (Lema complementului Schur) Dacă A este o matrice simetrică pozitiv-definită şi A_k este o submatrice principală de ordinul $k \times k$ a lui A, atunci complementul Schur al lui A, în raport cu A_k, este simetric şi pozitiv-definit.

Demonstraţie. Complementul S este simetric pe baza exerciţiului 31.1-7. Rămâne să arătăm că S este pozitiv-definit. Considerăm partiţionarea lui A dată în ecuaţia (31.28).

Pentru orice vector x nenul, avem $x^{\mathrm{T}}Ax > 0$ prin ipoteză. Fie x împărţit în doi subvectori y şi z, compatibili cu A_k şi respectiv cu C. Întrucât A_k^{-1} există, avem

$$\begin{aligned} x^{\mathrm{T}}Ax &= (y^{\mathrm{T}}\ z^{\mathrm{T}})\begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & c \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix} \\ &= y^{\mathrm{T}}A_k y + y^{\mathrm{T}}B^{\mathrm{T}}z + z^{\mathrm{T}}By + x^{\mathrm{T}}Cz \\ &= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}}A_k(y + A_k^{-1}B^{\mathrm{T}}z) + z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z, \end{aligned} \tag{31.30}$$

(se va verifica prin înmulţire). Această ultimă relaţie se echivalează cu "completarea pătratelor" formei pătratice (vezi exerciţiul 31.6-2.)

Întrucât avem $x^{\mathrm{T}}Ax > 0$ pentru orice x nenul, fixăm un z nenul şi apoi îl alegem pe y astfel încât $y = -A_k^{-1}B^{\mathrm{T}}z$, care anulează primul termen din relaţia (31.30), obţinându-se

$$z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z = z^{\mathrm{T}}Sz$$

ca valoare a expresiei. Pentru orice $z \neq 0$, avem $z^{\mathrm{T}}Sz = x^{\mathrm{T}}Ax > 0$, adică S este pozitiv-definit. ■

Corolarul 31.16 Descompunerea LU a unei matrice simetrice pozitiv-definite nu implică niciodată împărţirea cu zero.

Demonstraţie. Fie A o matrice simetrică pozitiv-definită. Vom demonstra o afirmaţie mai tare decât enunţul corolarului: orice pivot este strict pozitiv. Primul pivot este a_{11}. Fie e_1 primul vector unitate prin care se obţine $a_{11} = e_1^{\mathrm{T}}Ae_1 > 0$. Întrucât primul pas al descompunerii LU produce complementul Schur al lui A în raport cu $A_1 = (a_{11})$, lema 31.15 implică faptul că toţi pivoţii sunt pozitivi prin inducţie. ■

Aproximarea prin metoda celor mai mici pătrate

Ajustarea curbelor printr-o mulţime dată de puncte este o aplicaţie importantă a matricelor simetrice pozitiv-definite. Presupunem că se dă o mulţime de m puncte

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m),$$

unde y_i sunt cu erori de măsurare. Am dori să determinăm o funcţie $F(x)$, astfel încât

$$y_i = F(x_i) + \eta_i \tag{31.31}$$

pentru $i = 1, 2, \ldots, m$, unde erorile de aproximare η_i sunt mici. Forma funcţiei F depinde de problemă. Aici, presupunem că are forma unei sume liniare ponderate,

$$F(x) = \sum_{j=1}^{n} c_j f_j(x),$$

unde numărul n al termenilor şi ***funcţiile de bază*** f_j specifice se aleg ţinându-se seama de problemă. O alegere frecventă este $f_j(x) = x^{j-1}$, ceea ce înseamnă că

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1},$$

este un polinom în x, de gradul $n - 1$.

Alegând $n = m$, putem calcula *exact* fiecare y_i din ecuaţia (31.31).

O astfel de funcţie F de grad înalt "aproximează perturbaţiile" la fel de bine ca şi datele, dar, cu toate acestea, în general, dă rezultate slabe când se foloseşte la determinarea lui y, pentru valori ale lui x, care nu sunt prevăzute în prealabil. De obicei, este mai bine să se aleagă n semnificativ mai mic decât m şi să se spere că, alegând bine coeficienţii c_j, se va găsi o funcţie F care să aproximeze bine punctele. Anumite principii teoretice există pentru a-l alege pe n, dar ele sunt în afara scopului acestui text. În orice caz, o dată ales n, ajungem în final la un sistem de ecuaţii supradefinit, pe care dorim să-l rezolvăm cât mai bine cu putinţă. Arătăm cum se poate face acest lucru.

Fie

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix}$$

matricea valorilor funcţiilor de bază în punctele date; adică $a_{ij} = f_j(x_i)$. Fie $c = (c_k)$ vectorul n-dimensional al coeficienţilor pe care vrem să-l determinăm. Atunci,

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

este vectorul m-dimensional de "valori previzibile" pentru y. Astfel,

$$\eta = Ac - y$$

este vectorul m-dimensional de ***erori de aproximare***.

Pentru a minimiza erorile de aproximare, trebuie să minimizăm norma vectorului eroare η, ceea ce are ca ***soluţie cele mai mici pătrate***, deoarece

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2}.$$

Întrucât

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2 ,$$

putem minimiza $\|\eta\|$ prin diferenţierea lui $\|\eta\|^2$ în raport cu fiecare c_k şi egalând rezultatul cu zero:

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2 \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right) a_{ik} = 0. \tag{31.32}$$

Cele n ecuaţii (31.32) pentru $k = 1, 2, \ldots, n$ sunt echivalente cu o singură ecuaţie matriceală

$$(Ac - y)^{\mathrm{T}} A = 0$$

care este echivalentă cu

$$A^{\mathrm{T}}(Ac - y) = 0,$$

(folosind exerciţiul 31.1-3), care implică

$$A^{\mathrm{T}} Ac = A^{\mathrm{T}} y. \tag{31.33}$$

În statistică, aceasta se numeşte ***ecuaţie normală***. Matricea $A^{\mathrm{T}}A$ este simetrică pe baza exerciţiului 31.1-3, iar dacă A are un rang coloană complet, atunci $A^{\mathrm{T}}A$ este pozitiv-definită. Deci, $(A^{\mathrm{T}}A)^{-1}$ există, iar soluţia ecuaţiei (31.33) este

$$c = ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})y = A^{+}y, \tag{31.34}$$

unde matricea $A^{+} = ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})$ se numeşte ***pseudoinversa*** matricei A. Pseudoinversa este o generalizare naturală a noţiunii de inversă a unei matrice pentru cazul în care A nu este pătratică. (Să se compare ecuaţia (31.34), ca soluţie aproximativă a lui $Ac = y$, cu soluţia $A^{-1}b$ care este soluţia exactă a sistemului $Ax = b$.)

Ca un exemplu de folosire a aproximării prin metoda celor mai mici pătrate, să presupunem că avem 5 puncte

$$(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3)$$

indicate prin puncte negre în figura 31.3. Dorim să aproximăm aceste puncte printr-un polinom de gradul doi:

$$F(x) = c_1 + c_2 x + c_3 x^2.$$

Începem cu matricea valorilor funcţiei de bază

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix}.$$

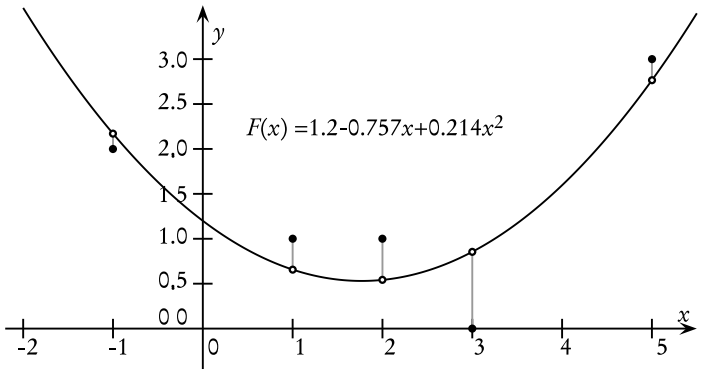


Figura 31.3 Aproximarea cu un polinom de gradul doi pe mulţimea de puncte $(-1, 2)$, $(1, 1)$, $(2, 1)$, $(3, 0)$, $(5, 3)$ prin metoda celor mai mici pătrate. Punctele negre sunt punctele date, iar punctele albe sunt valorile estimate ale acestora prin polinomul $F(x) = 1.2 - 0.757x + 0.214x^2$, polinom de gradul doi care minimizează suma pătratelor erorilor. Eroarea pentru fiecare punct dat este vizualizată haşurat.

Pseudoinversa este

$$A^+ = \begin{pmatrix} 0,500 & 0,300 & 0,200 & 0,100 & -0,100 \\ -0,388 & 0,093 & 0,190 & 0,193 & -0,088 \\ 0,060 & -0,036 & -0,048 & -0,036 & 0,060 \end{pmatrix}.$$

Înmulţind pe y cu A^+, obţinem vectorul coeficient

$$c = \begin{pmatrix} 1,200 \\ -0,757 \\ 0,214 \end{pmatrix},$$

deci, polinomul de gradul doi va fi:

$$F(x) = 1,200 - 0,757x + 0,214x^2$$

care este cea mai apropiată aproximare cu polinoame de gradul doi pentru valorile date, în sensul celor mai mici pătrate.

Din motive practice, rezolvăm ecuaţiile normale (31.33) înmulţind pe y cu A^{T} şi, apoi, găsind descompunerea LU a lui $A^{\mathrm{T}}A$. Dacă A are rangul complet, se garantează faptul că $A^{\mathrm{T}}A$ este nesingulară, deoarece este simetrică şi pozitiv-definită. (Vezi exerciţiul 31.1-3 şi teorema 31.6.)

Exerciţii

31.6-1 Demonstraţi că orice element de pe diagonala unei matrice simetrice pozitiv-definite este pozitiv.

31.6-2 Fie $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ o matrice de ordinul 2×2 simetrică şi pozitiv definită. Demonstraţi că determinantul $ac - b^2$ este pozitiv prin "completarea pătrată" într-o manieră similară celei utilizate în demonstraţia lemei 31.15.

31.6-3 Demonstraţi că elementul maxim într-o matrice simetrică pozitiv-definită se află pe diagonală.

31.6-4 Demonstraţi că determinantul fiecărei submatrice principale a unei matrice simetrice pozitiv-definite este pozitiv.

31.6-5 Fie A_k submatricea principală k a unei matrice A, simetrice pozitiv-definite. Arătaţi că $\det(A_k)/\det(A_{k-1})$ este al k-lea pivot din descompunerea LU, unde prin convenţie $\det(A_0) = 1$.

31.6-6 Găsiţi o funcţie de forma:

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

care este cea mai bună aproximare prin metoda celor mai mici pătrate pentru punctele:

$$(1,1), (2,1), (3,3), (4,8).$$

31.6-7 Arătaţi că pseudoinversa A^+ satisface următoarele patru ecuaţii:

$$\begin{aligned} AA^+A &= A, \\ A^+AA^+ &= A^+, \\ (AA^+)^{\mathrm{T}} &= AA^+, \\ (A^+A)^{\mathrm{T}} &= A^+A. \end{aligned}$$

Probleme

31-1 Algoritmul lui Schur de înmulţire a matricelor booleene
În secţiunea 31.3, am observat că algoritmul lui Strassen de înmulţire a matricelor nu poate fi aplicat direct la înmulţirea matricelor booleene deoarece cvasiinelul boolean $Q = (\{0,1\}, \vee, \wedge, 0, 1)$ nu este un inel. Teorema 31.10 a arătat că, dacă folosim operaţii aritmetice asupra cuvintelor de $O(\lg n)$ biţi, am putea totuşi să aplicăm metoda lui Strassen, pentru a înmulţi matrice booleene de ordinul $n \times n$, într-un timp $O(n^{\lg 7})$. Cercetăm o metodă probabilistică care utilizează numai operaţii pe biţi care să atingă o limită aproape tot atât de bună şi cu şanse mici de eroare.

a. Arătaţi că $R = (\{0,1\}, \oplus, \wedge, 0, 1)$, unde $\oplus$ este operatorul XOR, este un inel.

Fie $A = (a_{ij})$ şi $B = (b_{ij})$ două matrice booleene de ordinul $n \times n$ şi fie $C = (c_{ij}) = AB$ în cvasiinelul Q. Generăm $A' = (a'_{ij})$ din A, folosind următoarea procedură de randomizare:

- Dacă $a_{ij} = 0$, atunci $a'_{ij} = 0$.
- Dacă $a_{ij} = 1$, atunci $a'_{ij} = 1$ cu probabilitatea $1/2$ şi $a'_{ij} = 0$ cu aceeaşi probabilitate $1/2$. Alegerile aleatoare pentru fiecare element sunt independente.

b. Fie $c' = (c'_{ij}) = A'B$ în inelul R. Arătaţi că $c_{ij} = 0$ implică $c'_{ij} = 0$. Arătaţi că $c_{ij} = 1$ implică $c'_{ij} = 1$ cu probabilitatea $1/2$.

c. Arătaţi că, pentru orice $\epsilon > 0$, probabilitatea ca un c'_{ij} dat să nu ia niciodată valoarea c_{ij} pentru $\lg(n^2/\epsilon)$ alegeri independente ale matricei A', este cel mult ϵ/n^2. Arătaţi că probabilitatea ca toţi c'_{ij} să ia valorile lor corecte, cel puţin o dată, este cel puţin $1 - \epsilon$.

d. Daţi un algoritm de randomizare cu timp de ordinul $O(n^{\lg 7} \lg n)$ care calculează produsul matricelor booleene ale cvasiinelului Q de ordinul $n \times n$, cu probabilitatea de cel puţin $1 - 1/n^k$ pentru orice constantă $k > 0$. Singurele operaţii permise asupra elementelor matricelor sunt $\vee, \wedge$ şi $\oplus$.

31-2 Sisteme de ecuaţii liniare tridiagonale

Considerăm matricea tridiagonală

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}$$

a. Găsiţi o descompunere LU a lui A.

b. Rezolvaţi ecuaţia $Ax = (1 \quad 1 \quad 1 \quad 1 \quad 1)^{\mathrm{T}}$ utilizând substituţiile înainte şi înapoi.

c. Găsiţi inversa lui A.

d. Arătaţi că, pentru orice matrice simetrică pozitiv-definită, tridiagonală de ordinul $n \times n$ şi orice vector b, n-dimensional, ecuaţia $Ax = b$ poate fi rezolvată într-un timp de ordinul $O(n)$ prin descompunere LU. Argumentaţi că orice metodă bazată pe determinarea lui A^{-1} este, în cel mai defavorabil caz, asimptotic mai costisitoare.

e. Arătaţi că, pentru orice matrice A de ordinul $n \times n$ nesingulară, tridiagonală şi pentru orice vector b, n-dimensional, ecuaţia $Ax = b$ poate fi rezolvată într-un timp de ordinul $O(n)$ prin descompunerea LUP.

31-3 Spline-uri

O metodă practică pentru a interpola o mulţime de puncte cu o curbă este aceea de a folosi ***spline-uri cubice***. Se dă o mulţime $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ de $n+1$ perechi de valori-puncte, unde $x_0 < x_1 < \ldots < x_n$. Dorim să interpolăm punctele printr-o curbă cubică $f(x)$ pe porţiuni (spline). Curba $f(x)$ este alcătuită din n polinoame cubice $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ pentru $i = 0, 1, \ldots, n-1$, unde pentru x în domeniul $x_i \le x \le x_{i+1}$, valoarea curbei este dată de $f(x) = f_i(x - x_i)$. Punctele x_i în care polinoamele cubice sunt "legate" se numesc ***noduri***. Pentru simplificare, vom presupune că $x_i = i$, pentru $i = 0, 1, \ldots, n$.

Pentru a asigura continuitatea lui $f(x)$, se cere ca

$$f(x_i) = f_i(0) = y_i, f(x_{i+1}) = f_i(1) = y_{i+1}$$

pentru $i = 0, 1, \ldots, n-1$. Pentru a asigura faptul ca $f(x)$ să fie suficient de netedă, vom impune ca derivata de ordinul întâi să fie continuă în fiecare nod:

$$f'(x_{i+1}) = f'_i(1) = f'_{i+1}(0)$$

pentru $i = 0, 1, \ldots, n-1$.

a. Să presupunem că, pentru $i = 0, 1, \ldots, n$, se dau nu numai perechile de valori-puncte (x_i, y_i) ci şi derivatele de ordinul întâi $D_i = f'(x_i)$ în fiecare nod. Exprimaţi fiecare coeficient a_i, b_i, c_i şi d_i în funcţie de valorile y_i, y_{i+1}, D_i şi D_{i+1}. (Amintim că $x_i = i$.) Stabiliţi cât de rapid pot fi calculaţi cei $4n$ coeficienţi din perechile valori-puncte şi derivatele de ordinul întâi?

Trebuie stabilit modul de alegere a derivatelor de ordinul întâi ale lui $f(x)$ în noduri. O metodă este aceea de a cere ca derivatele de ordinul doi să fie continue în noduri:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

pentru $i = 0, 1, \ldots, n-1$. În primul şi ultimul nod, presupunem că $f''(x_0) = f_0''(0) = 0$ şi $f''(x_n) = f_n''(1) = 0$; aceste presupuneri determină ca $f(x)$ să fie un spline cubic ***natural***.

b. Utilizaţi restricţiile de continuitate ale derivatei a doua pentru a arăta că pentru $i = 1, 2, \ldots, n-1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}). \tag{31.35}$$

c. Arătaţi că

$$2D_0 + D_1 = 3(y_1 - y_0), \tag{31.36}$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}). \tag{31.37}$$

d. Rescrieţi relaţiile (31.35)–(31.37) printr-o ecuaţie matriceală implicând vectorul de necunoscute $D = \langle D_0, D_1, \ldots, D_n \rangle$. Ce atribute are matricea din ecuaţia respectivă?

e. Argumentaţi faptul că o mulţime de $n+1$ perechi de valori-puncte poate fi interpolată cu un spline cubic natural într-un timp de ordinul $O(n)$ (vezi problema 31-2).

f. Arătaţi cum se determină un spline cubic natural care interpolează un set de $n+1$ puncte (x_i, y_i) care satisfac inegalităţile $x_0 < x_1 < \cdots < x_n$, chiar dacă x_i nu este în mod necesar egal cu i. Ce ecuaţie matriceală trebuie rezolvată şi cât de rapid se execută algoritmul respectiv?

Note bibliografice

Există multe texte excelente care descriu calculul numeric şi ştiinţific cu mai multe detalii decât am făcut-o noi aici. În special, ar merita citite următoarele: George şi Liu [81], Golub şi Van Loan [89], Press, Flannery, Teukolsky şi Vetterling [161, 162] şi Strang [181, 182].

Publicarea algoritmului lui Strassen în 1969 [183] a cauzat multă emoţie. Până atunci, s-a imaginat că algoritmul ar putea fi îmbunătăţit. Limita superioară asimptotică a dificultăţii înmulţirii matricelor a fost de atunci îmbunătăţită considerabil. Cel mai eficient algoritm asimptotic pentru înmulţirea matricelor de ordinul $n \times n$, dat de Coppersmith şi Winograd [52] se execută într-un timp de ordinul $O(n^{2.376})$. Reprezentarea grafică a algoritmului lui Strassen a

fost dată de Paterson [155]. Fischer şi Meyer [67] au adaptat algoritmul lui Strassen la matricele booleene (teorema 31.10).

Eliminarea gaussiană, pe care se bazează descompunerile LU şi LUP, a fost prima metodă sistematică de rezolvare a sistemelor de ecuaţii liniare. De asemenea, ea a fost printre primii algoritmi numerici. Deşi metoda a fost cunoscută mai de mult, descoperirea ei este atribuită lui C. F. Gauss(1777–1855). De asemenea, în faimoasa sa lucrare [183], Strassen a arătat că o matrice de ordinul $n \times n$ poate fi inversată într-un timp de ordinul $O(n^{\lg 7})$. Winograd [203] a demonstrat, originar că înmulţirea matricelor nu este mai grea decât inversarea lor, iar reciproca se datorează lui Aho, Hopcroft şi Ullman [4].

Strang [182] are o prezentare excelentă despre matricele simetrice pozitiv-definite şi despre algebra liniară în general. El a făcut următoarea remarcă la pagina 334: "Elevii clasei mele adesea mă întreabă despre matricele *nesimetrice* pozitiv-definite. Eu niciodată nu am folosit acest termen".

32 Polinoame şi TFR

Metoda directă de a aduna două polinoame de grad n necesită un timp $\Theta(n)$, dar metoda directă pentru înmulţire necesită un timp $\Theta(n^2)$. În acest capitol, vom arăta cum transformata Fourier rapidă sau TFR (Fast Fourier Transform în engleză, prescurtat FFT) poate reduce timpul de înmulţire a două polinoame la $\Theta(n \lg n)$.

Polinoame

Un ***polinom*** în variabila x peste câmpul (algebric) F este o funcţie $A(x)$ care poate fi reprezentată după cum urmează:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j.$$

Vom numi n ***limita de grad*** a polinomului, iar valorile $a_0, a_1, \ldots, a_{n-1}$ le vom numi ***coeficienţii*** polinomului. Coeficienţii sunt elemente ale câmpului F, de regulă corpul $\mathbb{C}$ al numerelor complexe. Se spune că un polinom $A(x)$ are ***gradul*** k dacă k este rangul cel mai mare pentru care coeficientul a_k este nenul. Gradul unui polinom cu limita de grad n poate fi orice întreg din intervalul $[0, n-1]$. Reciproc, un polinom de grad k este un polinom cu limita de grad n, pentru orice $n > k$.

Există o varietate de operaţii pe care dorim să le definim (şi) pentru polinoame. Pentru ***adunarea a două polinoame***, dacă $A(x)$ şi $B(x)$ sunt polinoame cu limita de grad n, spunem că ***suma*** lor este un polinom $C(x)$, tot cu limita de grad n, astfel încât $C(x) = A(x) + B(x)$, pentru orice x din câmpul peste care s-au definit polinoamele. Respectiv, dacă

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

şi

$$B(x) = \sum_{j=0}^{n-1} b_j x^j,$$

atunci

$$C(x) = \sum_{j=0}^{n-1} c_j x^j,$$

unde $c_j = a_j + b_j$ pentru $j = 0, 1, \ldots, n-1$. De exemplu, dacă $A(x) = 6x^3 + 7x^2 - 10x + 9$ şi $B(x) = -2x^3 + 4x - 5$, atunci $C(x) = 4x^3 + 7x^2 - 6x + 4$.

Pentru ***înmulţirea a două polinoame***, dacă $A(x)$ şi $B(x)$ sunt polinoame cu limita de grad n, vom spune că ***produsul*** lor $C(x)$ este un polinom cu limita de grad $2n-1$, astfel încât $C(x) = A(x)B(x)$, pentru orice x din câmpul peste care sunt definite polinoamele. Probabil că aţi înmulţit deja polinoame, înmulţind fiecare termen din $A(x)$ cu fiecare termen din $B(x)$ şi

reducând termenii cu puteri egale. De exemplu, putem înmulţi polinoamele $A(x) = 6x^3 + 7x^2 - 10x + 9$ şi $B(x) = -2x^3 + 4x - 5$ după cum urmează:

$$\begin{array}{rrrrrrr}
 & & & 6x^3 & +7x^2 & -10x & +9 \\
 & & & -2x^3 & & +4x & -5 \\
\hline
 & & & -30x^3 & -35x^2 & +50x & -45 \\
 & & 24x^4 & +28x^3 & -40x^2 & +36x & \\
-12x^6 & -14x^5 & +20x^4 & -18x^3 & & & \\
\hline
-12x^6 & -14x^5 & +44x^4 & -20x^3 & -75x^2 & +86x & -45
\end{array}$$

Un alt mod de a exprima produsul $C(x)$ este

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \tag{32.1}$$

unde

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}. \tag{32.2}$$

De notat că grad(C) = grad(A) + grad(B), ceea ce implică

$$\begin{aligned}
\text{limita_de_grad(C)} &= \text{limita_de_grad(A)} + \text{limita_de_grad(B)} - 1 \leq \\
&\leq \text{limita_de_grad(A)} + \text{limita_de_grad(B)}
\end{aligned}$$

Vom vorbi, totuşi, despre limita de grad a lui C ca fiind suma limitelor de grad ale lui A şi B, deoarece un polinom care are limita de grad k are, de asemenea, limita de grad $k + 1$.

Rezumatul capitolului

Secţiunea 32.1 prezintă două moduri de a reprezenta polinoamele: reprezentarea prin coeficienţi şi reprezentarea prin valori pe puncte. Metoda directă de înmulţire a polinoamelor – ecuaţiile (32.1) şi (32.2) – necesită un timp $\Theta(n^2)$ când polinoamele sunt reprezentate prin coeficienţi, dar numai un timp $\Theta(n)$ când ele sunt reprezentate sub formă de valori pe puncte. Putem, totuşi, să înmulţim polinoamele, utilizând forma de reprezentare prin coeficienţi în timp $\Theta(n \lg n)$ făcând conversia între cele două reprezentări. Pentru a vedea cum funcţionează această idee, trebuie să studiem mai întâi rădăcinile complexe ale unităţii, prezentat în secţiunea 32.2. Apoi, pentru a realiza conversia, utilizăm transformata Fourier rapidă şi inversa sa, descrise, de asemenea, în secţiunea 32.2. În secţiunea 32.3 se prezintă o implementare eficientă a transformatei Fourier rapide, atât în modelul serial, cât şi în cel paralel.

Acest capitol utilizează intensiv numerele complexe, iar simbolul i va fi utilizat exclusiv pentru a desemna valoarea $\sqrt{-1}$.

32.1. Reprezentarea polinoamelor

Reprezentările polinoamelor prin coeficienţi şi prin valori pe puncte sunt, într-un anume sens, echivalente; adică, un polinom reprezentat sub formă de valori are un corespondent unic în

reprezentarea sub formă de coeficienţi. În această secţiune, vom introduce cele două reprezentări şi vom arăta cum se pot ele combina pentru a permite înmulţirea a două polinoame cu limita de grad n în timp $\Theta(n \lg n)$.

Reprezentarea prin coeficienţi

O ***reprezentare prin coeficienţi*** a unui polinom $A(x) = \sum_{j=0}^{n-1} a_j x^j$ cu limita de grad n este un vector de coeficienţi $a = (a_0, a_1, \ldots, a_{n-1})$. În ecuaţiile matriceale din acest capitol, vom trata, în general, aceşti vectori ca vectori coloană.

Reprezentarea prin coeficienţi este convenabilă pentru anumite operaţii cu polinoame. De exemplu, operaţia de ***evaluare*** a polinomului $A(x)$, într-un punct dat x_0, constă în calculul valorii $A(x_0)$. Evaluarea, utilizând ***regula (schema) lui Horner***, consumă un timp $\Theta(n)$:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\ldots)).$$

Similar, adunarea a două polinoame, reprezentate sub forma vectorilor de coeficienţi $a = (a_0, a_1, \ldots, a_{n-1})$ şi $b = (b_0, b_1, \ldots, b_{n-1})$, consumă un timp $\Theta(n)$: trebuie, doar, să listăm vectorul $c = (c_0, c_1, \ldots, c_{n-1})$, unde $c_j = a_j + b_j$, pentru $j = 0, 1, \ldots, n-1$.

Să considerăm acum înmulţirea a două polinoame cu limita de grad n, $A(x)$ şi $B(x)$, reprezentate sub formă de coeficienţi. Dacă utilizăm metoda descrisă prin ecuaţiile (32.1) şi (32.2), înmulţirea a două polinoame necesită un timp $\Theta(n^2)$ deoarece fiecare coeficient din vectorul a trebuie înmulţit cu fiecare coeficient din vectorul b. Operaţia de înmulţire a două polinoame reprezentate sub formă de coeficienţi, pare să fie considerabil mai dificilă decât evaluarea unui polinom sau adunarea a două polinoame. Vectorul de coeficienţi rezultat c, dat de ecuaţia (32.2), se mai numeşte şi ***convoluţia*** vectorilor de intrare a şi b şi se notează cu $c = a \otimes b$. Deoarece înmulţirea polinoamelor şi calculul convoluţiilor sunt probleme de calcul de o importanţă practică fundamentală, acest capitol se concentrează asupra algoritmilor eficienţi pentru rezolvarea lor.

Reprezentarea prin valori pe puncte

O ***reprezentare prin valori pe puncte*** a unui polinom $A(x)$, cu limita de grad n, este o mulţime de n ***puncte*** (perechi de valori)

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

astfel încât toate valorile x_k sunt distincte şi

$$y_k = A(x_k) \tag{32.3}$$

pentru $k = 0, 1, \ldots, n-1$. Un polinom poate avea mai multe reprezentări prin valori pe puncte deoarece orice mulţime de n puncte distincte $x_0, x_1, \ldots, x_{n-1}$ poate fi utilizată ca bază pentru reprezentare.

Calculul unei reprezentări prin valori pe puncte pentru un polinom dat sub formă de coeficienţi este, în principiu, simplă deoarece tot ce avem de făcut este să alegem n puncte distincte $x_0, x_1, \ldots, x_{n-1}$ şi, apoi, să evaluăm $A(x_k)$ pentru $k = 0, 1, \ldots, n-1$. Cu schema lui Horner, aceste evaluări în n puncte consumă un timp $\Theta(n^2)$. Vom vedea, mai târziu, că, dacă alegem x_k în mod inteligent, acest calcul poate fi accelerat până la un timp $\Theta(n \lg n)$.

Operaţia inversă evaluării – determinarea reprezentării sub formă de coeficienţi dintr-o reprezentare sub formă de valori pe puncte – se numeşte ***interpolare***. Teorema următoare ne

arată că interpolarea este bine definită în ipoteza că limita de grad a polinomului de interpolare este egală cu numărul de puncte.

Teorema 32.1 (Unicitatea polinomului de interpolare) Pentru orice mulţime de n puncte $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ există un polinom unic $A(x)$ cu limita de grad n, astfel încât, $y_k = A(x_k)$ pentru $k = 0, 1, \ldots, n-1$.

Demonstraţie. Demonstraţia se bazează pe existenţa inversei unei anumite matrice. Ecuaţia (32.3) este echivalentă cu ecuaţia matriceală

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \tag{32.4}$$

Matricea din stânga se notează cu $V(x_0, x_1, \ldots, x_{n-1})$ şi este cunoscută sub numele de matrice Vandermonde. Conform exerciţiului 31.1-10, această matrice are determinantul

$$\prod_{j<k} (x_k - x_j),$$

şi, de aceea, conform teoremei 31.5, dacă x_k sunt distincte, ea este inversabilă (adică nesingulară). Astfel, coeficienţii a_j pot fi determinaţi unic din reprezentarea prin puncte cu

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y.$$

■

Demonstraţia teoremei 32.1 descrie un algoritm de interpolare bazat pe rezolvarea sistemului de ecuaţii liniare (32.4). Utilizând algoritmul de descompunere LU, din capitolul 31, putem rezolva acest sistem în timp $O(n^3)$.

Un algoritm mai rapid pentru interpolare pe n puncte se bazează pe ***formula lui Lagrange***:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}. \tag{32.5}$$

Este uşor de verificat că membrul drept al ecuaţiei (32.5) este un polinom cu limita de grad n care satisface $A(x_k) = y_k$ pentru $k = 0, 1, \ldots, n-1$. Exerciţiul 32.1-4 pune problema calculării coeficienţilor lui A în timp $\Theta(n^2)$, utilizând formula lui Lagrange.

Astfel, evaluarea în n puncte şi interpolarea sunt operaţii bine definite şi una este inversa celeilalte; ele convertesc reprezentarea prin coeficienţi a unui polinom în reprezentare prin valori pe puncte şi invers.[1] Algoritmul descris mai sus, pentru aceste probleme, durează un timp $\Theta(n^2)$.

[1] Este un fapt notoriu că interpolarea este o problemă instabilă din punct de vedere numeric. Deşi abordarea de aici este corectă din punct de vedere matematic, diferenţe mici în valorile de intrare sau erorile de rotunjire în timpul calculelor pot cauza diferenţe mari în rezultat.

Această reprezentare prin valori pe puncte este convenabilă pentru multe operaţii cu polinoame. Pentru adunare, dacă $C(x) = A(x) + B(x)$, atunci $C(x_k) = A(x_k) + B(x_k)$ pentru orice punct x_k. Mai precis, dacă avem o reprezentare prin valori pe puncte pentru A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

şi una pentru B

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

(de notat că A şi B sunt evaluate în *aceleaşi* n puncte), atunci o reprezentare prin valori pe puncte pentru C este

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\}.$$

Timpul necesar pentru a aduna două polinoame cu limita de grad n, reprezentate sub formă de valori pe puncte, este $\Theta(n)$.

De asemenea, reprezentarea prin valori pe puncte este convenabilă pentru înmulţirea polinoamelor. Dacă $C(x) = A(x)B(x)$, atunci $C(x_k) = A(x_k)B(x_k)$ pentru orice x_k şi, astfel, putem înmulţi punct cu punct reprezentarea prin valori pe puncte a lui A cu reprezentarea prin valori pe puncte a lui B pentru a obţine reprezentarea prin valori pe puncte a lui C. Trebuie, totuşi, să facem faţă problemei că limita de grad a lui C este suma limitelor de grad ale lui A şi B. O reprezentare standard prin valori pe puncte pentru A şi B constă din n puncte, dar deoarece limita de grad pentru C este $2n$, teorema 32.1 implică faptul că avem nevoie de $2n$ puncte pentru o reprezentare prin valori pe puncte a lui C. De aceea, trebuie să începem cu o reprezentare "extinsă" prin valori pe puncte a lui A şi B, constând fiecare, din $2n$ puncte. Dându-se o reprezentare extinsă prin valori pe puncte pentru A

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\}$$

şi o reprezentare extinsă prin valori pe puncte pentru B

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\},$$

atunci o reprezentare prin valori pe puncte pentru C este

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}.$$

Dându-se, la intrare, două polinoame în forma valorilor pe puncte, observăm că timpul necesar pentru a obţine forma prin valori pe puncte a rezultatului este $\Theta(n)$, mai mic decât timpul necesar pentru a înmulţi polinoamele reprezentate sub formă de coeficienţi.

În final, vom vedea cum putem să evaluăm un polinom dat sub forma valorilor pe puncte într-un nou punct. Pentru această problemă nu există, în aparenţă, nici o abordare mai simplă decât convertirea polinomului sub formă de coeficienţi şi apoi evaluarea sa în acel nou punct.

Înmulţirea rapidă a polinoamelor sub formă de coeficienţi

Oare putem utiliza metoda de înmulţire în timp liniar a polinoamelor reprezentate sub formă de valori pe puncte pentru a accelera înmulţirea polinoamelor reprezentate sub formă de coeficienţi? Răspunsul depinde de abilitatea noastră, de a converti rapid o reprezentare a

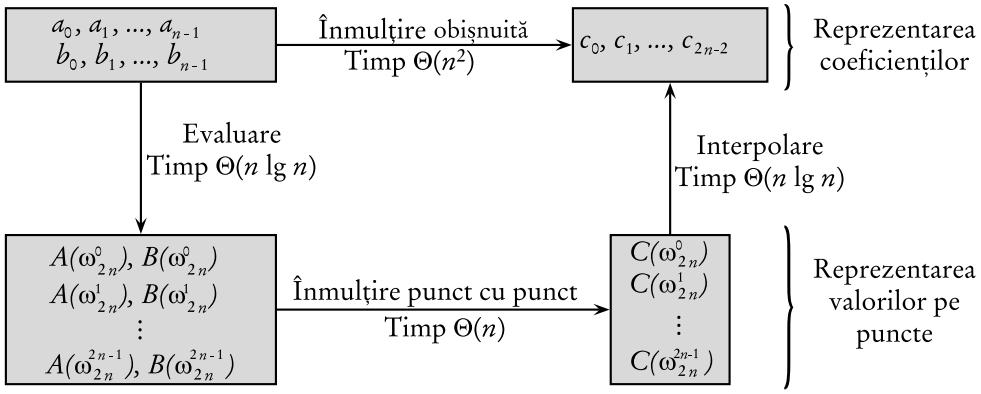


Figura 32.1 O schiţă grafică a procesului eficient de înmulţire a polinoamelor. Reprezentările din partea de sus sunt sub formă de coeficienţi, în timp ce cele din partea de jos sunt sub formă de valori pe puncte. Săgeţile de la stânga la dreapta corespund operaţiilor de înmulţire. Termenii ω_{2n} sunt rădăcinile de ordinul $2n$ ale unităţii.

unui polinom sub formă de coeficienţi, într-o reprezentare a unui polinom sub formă de valori pe puncte (evaluare) şi invers (interpolare).

Putem utiliza orice punct în care dorim să se facă evaluarea, dar, alegând punctele de evaluare cu grijă, putem face conversia între reprezentări într-un timp de numai $\Theta(n \lg n)$. Aşa cum vom vedea în secţiunea 32.2, dacă alegem "rădăcinile complexe ale unităţii" ca puncte de evaluare, putem genera o reprezentare prin valori pe puncte calculând transformata Fourier discretă (sau TFD[2]) a vectorului coeficienţilor. În secţiunea 32.2 se va arăta cum TFR realizează TFD şi inversa TFD în timp $\Theta(n \lg n)$.

Figura 32.1 ilustrează grafic această strategie. Un detaliu minor se referă la limita de grad. Produsul a două polinoame cu limita de grad n este un polinom cu limita de grad $2n$. Înainte de evaluarea polinoamelor de intrare A şi B, vom dubla limita de grad la $2n$, adăugând n coeficienţi 0 în extremitatea dreaptă a vectorului. Deoarece vectorii au $2n$ elemente, vom utiliza rădăcinile complexe de ordinul $2n$ ale unităţii pe care le vom nota în figura 32.1 cu ω_{2n}.

Dându-se TFR, avem următoarea procedură în timp $\Theta(n \lg n)$ pentru înmulţirea a două polinoame $A(x)$ şi $B(x)$ cu limita de grad n, unde intrarea şi ieşirea sunt reprezentate sub formă de coeficienţi. Presupunem că n este o putere a lui 2; această cerinţă poate fi satisfăcută întotdeauna, adăugând coeficienţi zero de rang mare.

1. *Dublarea limitei de grad*: creăm reprezentările sub formă de coeficienţi pentru $A(x)$ şi $B(x)$ ca polinoame cu limita de grad $2n$, completând cu n coeficienţi 0 de ordin mare pe fiecare din ele.

2. *Evaluare*: calcularea reprezentărilor sub formă de valori pe puncte ale lui $A(x)$ şi $B(X)$ de lungime $2n$ prin două aplicări ale TFR de ordin $2n$. Aceste reprezentări conţin valorile celor două polinoame în cele $2n$ rădăcini de ordinul $2n$ ale unităţii.

3. *Înmulţirea punctuală*: se calculează o reprezentare prin valori pe puncte pentru polinomul

[2] În engleză Discrete Fourier Transform (DFT)–n.t.

$C(x) = A(x)B(x)$, înmulţind aceste valori punct cu punct. Această reprezentare conţine valorile lui $C(x)$ în rădăcinile de ordinul $2n$ ale unităţii.

4. *Interpolare*: se creează reprezentarea prin coeficienţi a polinomului $C(x)$ printr-o singură aplicare a TFR pe $2n$ puncte pentru a calcula inversa TFD.

Paşii (1) şi (3) necesită un timp $\Theta(n)$, iar paşii (2) şi (4) necesită un timp $\Theta(n \lg n)$. Astfel, o dată ce am arătat cum se realizează TFR, am demonstrat rezultatul următor:

Teorema 32.2 Produsul a două polinoame cu limita de grad n poate fi calculat în timp $\Theta(n \lg n)$, cu polinoamele de intrare şi ieşire reprezentate sub formă de coeficienţi.

■

Exerciţii

32.1-1 Înmulţiţi polinoamele $A(x) = 7x^3 - x^2 + x - 10$ şi $B(x) = 8x^3 - 6x + 3$ utilizând ecuaţiile (32.1) şi (32.2).

32.1-2 Evaluarea unui polinom $A(x)$ cu limita de grad n într-un punct dat x_0 se poate realiza, de asemenea, împărţind $A(x)$ cu polinomul $(x - x_0)$ pentru a obţine un polinom $q(x)$ cu limita de grad $n - 1$ şi un rest r, astfel încât

$$A(x) = q(x)(x - x_0) + r.$$

Evident, $A(x_0) = r$. Arătaţi cum se poate calcula restul r şi coeficienţii lui $q(x)$ în timp $\Theta(n)$, date fiind x_0 şi coeficienţii lui A.

32.1-3 Obţineţi o reprezentare prin listă de valori pe puncte pentru $A^{rev}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$ dintr-o reprezentare prin valori pe puncte pentru $A(x) = \sum_{j=0}^{n-1} a_j x^j$, presupunând că nici unul din puncte nu este 0.

32.1-4 Arătaţi cum se poate utiliza ecuaţia (32.5) pentru a realiza interpolarea în timp $\Theta(n^2)$. (*Indicaţie:* Se calculează întâi $\prod_j (x - x_j)$ şi apoi se împarte cu $(x - x_k)$ după cum este necesar pentru numărătorul fiecărui termen. Vezi exerciţiul 32.1-1.)

32.1-5 Explicaţi ce este greşit în abordarea "evidentă" prin împărţire de polinoame, utilizând o reprezentare prin valori pe puncte. Discutaţi separat cazurile când acesta funcţionează corect şi când nu.

32.1-6 Considerăm două mulţimi A şi B, având fiecare n elemente întregi în intervalul de la 0 la $10n$. Dorim să calculăm ***suma cartezianǎ*** a lui A şi B definită prin

$$C = \{x + y : x \in A \text{ şi } y \in B\}.$$

De notat că numerele întregi din C sunt din intervalul $[0, 20n]$. Dorim să găsim elementele lui C şi numărul de posibilităţi în care fiecare element din C se realizează ca o sumă a elementelor din A şi B. Arătaţi că această problemă poate fi rezolvată în timp $O(n \lg n)$. (*Indicaţie:* Reprezentaţi A şi B ca polinoame de grad $10n$.)

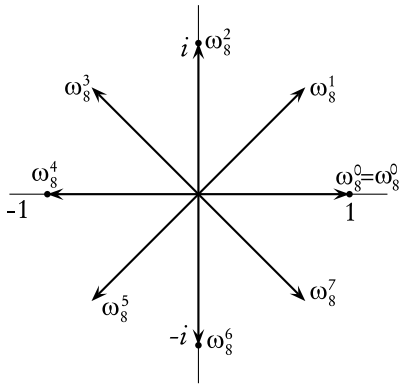


Figura 32.2 Valorile ω_8^0, ω_8^1, ..., ω_8^7 în planul complex, unde $\omega_8 = e^{2\pi i/8}$ este rădăcina complexă de ordinul 8 a unităţii.

32.2. TFD şi TFR

În secţiunea 32.1 am afirmat că, dacă utilizăm rădăcinile complexe ale unităţii, putem să evaluăm şi să interpolăm în timp $\Theta(n \lg n)$. În această secţiune definim rădăcinile complexe ale unităţii şi studiem proprietăţile lor, definim TFD şi apoi arătăm cum TFR calculează TFD şi inversa acesteia în timp $\Theta(n \lg n)$.

Rădăcinile complexe ale unităţii

O ***rădăcină complexă de ordinul n a unităţii*** este un număr complex ω cu proprietatea

$$\omega^n = 1.$$

Există exact n rădăcini complexe de ordinul n ale unităţii; acestea sunt $e^{2\pi ik/n}$ pentru $k = 0, 1, \ldots, n-1$. Pentru a interpreta această formulă, vom utiliza exponenţiala cu argument complex

$$e^{iu} = \cos(u) + i \sin(u).$$

Figura 32.2 ne arată că cele n rădăcini complexe ale unităţii sunt egal spaţiate pe cercul unitate. Valoarea

$$\omega_n = e^{2\pi i/n} \tag{32.6}$$

se numeşte ***rădăcină principală de ordinul n a unităţii***; toate celelalte rădăcini complexe de ordinul n ale unităţii sunt puteri ale lui ω_n. Cele n rădăcini de ordinul n ale unităţii

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1},$$

formează un grup în raport cu operaţia de înmulţire (vezi secţiunea 33.3). Acest grup are aceeaşi structură[3] ca grupul aditiv modulo n $(\mathbb{Z}_n, +)$, căci $\omega_n^n = \omega_n^0 = 1$ implică $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. La fel, $\omega_n^{-1} = \omega_n^{n-1}$. Proprietăţile esenţiale ale rădăcinilor de ordinul n ale unităţii sunt date de următoarele leme.

[3] Cele două grupuri sunt izomorfe – n.t.

Lema 32.3 (Lema de anulare) Pentru orice întregi $n \geq 0$, $k \geq 0$ şi $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k. \tag{32.7}$$

Demonstraţie. Lema rezultă direct din ecuaţia (32.6), deoarece

$$\omega_{dn}^{dk} = (e^{2\pi i/dn})^{dk} = (e^{2\pi i/n})^k = \omega_n^k.$$

■

Corolarul 32.4 Pentru orice întreg par $n > 0$

$$\omega_n^{n/2} = \omega_2 = -1.$$

Demonstraţie. Demonstraţia rămâne ca exerciţiu (exerciţiul 32.2-1). ■

Lema 32.5 (Lema de înjumătăţire) Dacă $n > 0$ este par, atunci pătratele celor n rădăcini complexe de ordinul n ale unităţii sunt cele $n/2$ rădăcini complexe de ordinul $n/2$ ale unităţii.

Demonstraţie. Din lema de anulare rezultă că $(\omega_n^k)^2 = \omega_{n/2}^k$, pentru orice întreg nenegativ k. De notat că dacă ridicăm la pătrat toate rădăcinile complexe de ordinul n ale unităţii, atunci fiecare rădăcină complexă de ordinul $n/2$ a unităţii se obţine exact de două ori, deoarece

$$(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k} = (\omega_n^k)^2.$$

Astfel, ω_n^k şi $\omega_n^{k+n/2}$ au acelaşi păstrat. Această propritate poate fi demonstrată, de asemenea, utiliz ând corolarul 32.4, deoarece $\omega_n^{n/2} = -1$ implică $\omega_n^{k+n/2} = -\omega_n^k$ şi deci $\left(\omega_n^{k+n/2}\right)^2 = \left(\omega_n^k\right)^2$. ■

Aşa după cum vom vedea, lema de înjumătăţire este esenţială pentru abordarea de tip divide şi stăpâneşte a conversiei între reprezentarea polinoamelor prin coeficienţi şi cea prin valori pe puncte, deoarece ea garantează că subproblemele recursive au dimensiunea pe jumătate.

Lema 32.6 (Lema de însumare) Pentru orice întreg $n \geq 1$ şi orice întreg nenegativ k nedivizibil prin n,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.$$

Demonstraţie. Din ecuaţia (3.3), aplicată unor valori complexe, obţinem

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} = \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} = \frac{(1)^k - 1}{\omega_n^k - 1} = 0.$$

Cerinţa ca n să nu fie divizor al lui k ne asigură că numitorul este nenul, deoarece $\omega_n^k = 1$ numai dacă k este divizibil cu n. ■

TFD

Reamintim că dorim să evaluăm polinomul

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

cu limita de grad n în $\omega_n^0, \omega_n^1, \omega_n^2 \ldots, \omega_n^{n-1}$ (adică rădăcinile complexe de ordinul n ale unităţii).[4] Fără a restrânge generalitatea, presupunem că n este o putere a lui 2, deoarece o limită de grad poate fi întotdeauna mărită – putem adăuga după necesităţi coeficienţi 0 de rang înalt. Presupunem că A este dat sub formă de coeficienţi: $a = (a_0, a_1, \ldots, a_{n-1})$. Definim rezultatul y_k, pentru $k = 0, 1, \ldots, n-1$, prin

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}. \tag{32.8}$$

Vectorul $y = (y_0, y_1, \ldots, y_{n-1})$ este ***transformata Fourier discretă (TFD)*** a vectorului coeficienţilor $a = (a_0, a_1, \ldots, a_{n-1})$. Scriem, de asemenea, y=TFD$_n(a)$.

TFR

Utilizând o metodă cunoscută sub numele de ***transformata Fourier rapidă (TFR)***, care profită de avantajul proprietăţilor speciale ale rădăcinilor complexe ale unităţii, putem calcula TFD$_n(a)$ în timp $\Theta(n \lg n)$, spre deosebire de timpul $\Theta(n^2)$ necesitat de metoda directă. Metoda TFR utilizează o strategie divide şi stăpâneşte, folosind separat coeficienţii cu indice par şi separat pe cei cu indice impar ai lui $A(x)$, pentru a defini două noi polinoame $A^{[0]}(x)$ şi $A^{[1]}(x)$ cu limita de grad $n/2$:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}, \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}. \end{aligned}$$

Observăm că $A^{[0]}$ conţine toţi coeficienţii cu indice par ai lui A (reprezentarea binară a acestor indici se termină cu 0) şi $A^{[1]}$ conţine toţi coeficienţii cu indice impar (reprezentarea lor binară se termină cu 1). Urmează că

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2), \tag{32.9}$$

şi, astfel, problema evaluării lui $A(x)$ în $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$se reduce la:

1. evaluarea polinoamelor $A^{[0]}(x)$ şi $A^{[1]}(x)$ cu limita de grad $n/2$ în punctele

   $$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2, \tag{32.10}$$

   şi apoi

2. combinarea rezultatelor conform ecuaţiei (32.9).

[4] Lungimea n are acelaşi rol ca şi $2n$ în secţiunea 32.1, deoarece dublăm limita de grad a polinomului înainte de evaluare. În contextul înmulţirii polinoamelor lucrăm, din acest motiv, cu rădăcini complexe de ordinul $2n$ ale unităţii.

Conform lemei de înjumătăţire, lista de valori (32.10) nu este constituită din n valori distincte, ci din cele $n/2$ rădăcini complexe de ordinul $n/2$ ale unităţii şi fiecare rădăcină apărând de exact două ori. De aceea, polinoamele $A^{[0]}$ şi $A^{[1]}$ cu limita de grad $n/2$ sunt evaluate recursiv în cele $n/2$ rădăcini complexe de ordinul $n/2$ ale unităţii. Aceste subprobleme au exact aceeaşi formă ca şi problema generală, dar pe jumătatea dimensiunii originale. Am reuşit să descompunem calculul TFD_n a n elemente în calculul a două $\text{TFD}_{n/2}$ a câte $n/2$ elemente. Această descompunere este baza următorului algoritm recursiv pentru TFR, care calculează TFD a unui vector cu n elemente $a = (a_0, a_1, \ldots, a_{n-1})$, unde n este o putere a lui 2.

TFR-RECURSIVĂ(a)
1: $n \leftarrow lungime[a]$ ▷ n este o putere a lui 2
2: **dacă** $n = 1$ **atunci**
3: **returnează** a
4: $\omega_n \leftarrow e^{2\pi i/n}$
5: $\omega \leftarrow 1$
6: $a^{[0]} \leftarrow (a_0, a_2, \ldots, a_{n-2})$
7: $a^{[1]} \leftarrow (a_1, a_3, \ldots, a_{n-1})$
8: $y^{[0]} \leftarrow$ TFR-RECURSIVĂ($a^{[0]}$)
9: $y^{[1]} \leftarrow$ TFR-RECURSIVĂ($a^{[1]}$)
10: **pentru** $k = 0, n/2 - 1$ **execută**
11: $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
12: $y_{k+(n/2)} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
13: $\omega \leftarrow \omega\omega_n$
14: **returnează** y ▷ se presupune că y este vector coloană

Procedura TFR-RECURSIVĂ lucrează după cum urmează. Liniile 2–3 reprezintă baza recursivităţii; TFD a unui element este elementul însuşi, deoarece în acest caz

$$y_0 = a_0\omega_1^0 = a_0 \cdot 1 = a_0.$$

Liniile 6–7 definesc vectorii coeficienţilor pentru polinoamele $A^{[0]}$ şi $A^{[1]}$. Liniile 4, 5 şi 13 garantează că ω este actualizat corespunzător, astfel ca ori de câte ori se execută liniile 11–12, $\omega = \omega_n^k$. (Păstrarea valorii ω, de la o iteraţie la alta, economiseşte timp, comparativ cu calculul lui ω_n^k de fiecare dată în ciclul **pentru**). Liniile 8–9 realizează calculul recursiv al lui $\text{TFD}_{n/2}$, punând, pentru $k = 0, 1, \ldots, n/2 - 1$,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k), \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k), \end{aligned}$$

sau, deoarece $\omega_{n/2}^k = \omega_n^{2k}$, conform lemei de anulare,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}), \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}). \end{aligned}$$

Liniile 11–12 combină rezultatele calculelor recursive pentru $\text{TFD}_{n/2}$. Pentru $y_0, y_1, \ldots, y_{n/2-1}$ linia 11 ne furnizează

$$y_k = y_k^{[0]} + \omega_n^k y_k^{[1]} = A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) = A(\omega_n^k),$$

unde ultima egalitate a acestui raţionament rezultă din ecuaţia (32.9). Pentru $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, luând $k = 0, 1, \ldots, n/2 - 1$, linia 12 produce

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} = y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} = A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) = \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) = A(\omega_n^{k+(n/2)}). \end{aligned}$$

A doua egalitate rezultă din prima deoarece $\omega_n^{k+(n/2)} = -\omega_n^k$. A patra linie rezultă din a treia, deoarece $\omega_n^n = 1$ implică $\omega_n^{2k} = \omega_n^{2k+n}$. Ultima egalitate rezultă din ecuaţia (32.9). Astfel, vectorul y returnat de TFR-RECURSIVĂ este într-adevăr TFD a vectorului de intrare a.

Pentru a determina timpul de execuţie al procedurii TFR-RECURSIVĂ, observăm că, excluzând apelurile recursive propriu-zise, fiecare apel consumă un timp $\Theta(n)$, unde n este lungimea vectorului de intrare. Aşadar, recurenţa pentru timpul de execuţie este

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n).$$

Astfel, putem evalua un polinom cu limita de grad n în rădăcinile complexe de ordinul n ale unităţii în timp $\Theta(n \lg n)$, utilizând transformata Fourier rapidă.

Interpolare în rădăcini complexe ale unităţii

Vom încheia acum schema de înmulţire polinomială arătând cum putem să interpolăm în rădăcinile complexe ale unităţii printr-un polinom, ceea ce ne permite să facem conversia de la forma prin valori pe puncte la forma prin coeficienţi. Vom interpola, scriind TFD ca pe o ecuaţie matriceală şi căutând forma matricei inverse.

Din ecuaţia (32.4) rezultă că putem scrie TFD ca pe un produs de matrice $y = V_n a$, unde V_n este o matrice Vandermonde ce conţine puterile adecvate ale lui ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \ldots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \ldots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \ldots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \ldots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Intrarea (k, j) a lui V_n este ω_n^{kj} pentru $j, k = 0, 1, \ldots, n-1$, iar exponenţii intrărilor lui V_n formează o tabelă a înmulţirii.

Pentru a realiza operaţia inversă, pe care o scriem ca $a = \text{TFD}_n^{-1}(y)$, înmulţim y cu matricea V_n^{-1}, inversa lui V_n.

Teorema 32.7 Pentru $j, k = 0, 1, \ldots, n-1$, intrarea (j, k) a lui V_n^{-1} este ω_n^{-kj}/n.

Demonstraţie. Vom arăta că $V_n^{-1} V_n = I_n$, unde I_n este matricea identică $n \times n$. Considerăm intrarea (j, j') a lui $V_n^{-1} V_n$:

$$[V_n^{-1} V_n]_{jj'} = \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) = \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n.$$

Această sumă este egală cu 1 dacă $j' = j$ şi este 0 în caz contrar, conform lemei de însumare (lema 32.6). Observăm că ne bazăm pe inegalitatea $-(n-1) < j' - j < n-1$ şi pe faptul că $j' - j$ este divizibil cu n, pentru ca lema de însumare să se poată aplica. ∎

Dându-se matricea inversă V_n^{-1}, $\mathrm{TFD}_n^{-1}(y)$ se calculează cu

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{32.11}$$

pentru $j = 0, 1, \ldots, n-1$. Comparând ecuaţiile (32.8) şi (32.11), observăm că, modificând algoritmul TFR pentru a schimba rolurile lui a şi y, înlocuind ω_n cu ω_n^{-1} şi împărţind fiecare element al rezultatului cu n, calculăm inversa TFD (vezi exerciţiul 32.2-4). Astfel, şi TFD_n^{-1} poate fi calculată în timp $\Theta(n \lg n)$.

Prin urmare, utilizând TFR şi inversa TFR, putem transforma reprezentarea sub formă de coeficienţi a unui polinom cu limita de grad n în reprezentarea prin valori pe puncte şi invers, în timp $\Theta(n \lg n)$. În contextul înmulţirii polinomiale, am demonstrat următoarea teoremă.

Teorema 32.8 (Teorema de convoluţie) Pentru oricare doi vectori a şi b de lungime n, unde n este o putere a lui 2,

$$a \otimes b = \mathrm{TFD}_{2n}^{-1}(\mathrm{TFD}_{2n}(a) \cdot \mathrm{TFD}_{2n}(b)),$$

unde vectorii a şi b sunt completaţi cu zerouri până la lungimea $2n$, iar $\cdot$ desemnează produsul pe componente al vectorilor cu $2n$ elemente. ∎

Exerciţii

32.2-1 Demonstraţi corolarul 32.4.

32.2-2 Calculaţi TFD a vectorului $(0, 1, 2, 3)$.

32.2-3 Rezolvaţi exerciţiul 32.1-1, utilizând o schemă cu timpul $\Theta(n \lg n)$.

32.2-4 Scrieţi pseudocodul pentru a calcula TFD_n^{-1} în timp $\Theta(n \lg n)$.

32.2-5 Descrieţi generalizarea TFR în cazul când n este o putere a lui 3. Daţi o recurenţă pentru timpul de execuţie şi rezolvaţi recurenţa.

32.2-6 ⋆ Presupunem că în loc de a realiza TFR a n elemente peste câmpul numerelor complexe (unde n este par), utilizăm inelul $\mathbb{Z}_m$ al întregilor modulo m, unde $m = 2^{tn/2} + 1$ şi t este un întreg pozitiv arbitrar. Utilizaţi $\omega = 2^t$ modulo m în loc de ω_n ca rădăcină principală de ordinul n a unităţii. Demonstraţi că TFD şi inversa TFD sunt bine definite în acest sistem.

32.2-7 Fie lista de valori $z_0, z_1, \ldots, z_{n-1}$ (posibil cu repetiţii), arătaţi cum pot fi găsiţi coeficienţii polinomului $P(x)$ cu limita de grad n ce are ca zerouri numerele $z_0, z_1, \ldots, z_{n-1}$ (posibil cu repetiţii). Procedura se va executa în timp $O(n \lg^2 n)$. (*Indicaţie:* polinomul $P(x)$ are un zero în z_j dacă şi numai dacă $P(x)$ este multiplu al lui $(x - z_j)$.)

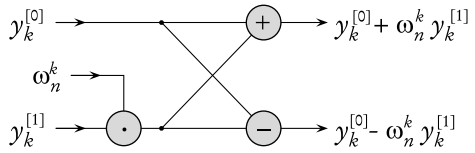


Figura 32.3 O operaţie fluture (butterfly). Cele două valori de intrare vin din stânga, ω_n^k este înmulţit cu $y_k^{[1]}$, iar suma şi diferenţa sunt ieşirile din dreapta. Figura poate fi interpretată ca un circuit combinaţional.

32.2-8 ⋆ Transformarea "cirip" (chirp transform) a unui vector $a = (a_0, a_1, \ldots, a_n)$ este vectorul $y = (y_0, y_1, \ldots, y_{n-1})$ unde $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ şi z este un număr complex. TFD este, deci, un caz particular al transformării "cirip", în cazurile în care $z = \omega_n$. Demonstraţi că transformarea "cirip" poate fi evaluată în timp $O(n \lg n)$ pentru orice număr complex z. (*Indicaţie:* utilizaţi ecuaţia

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right) \left(z^{-(k-j)^2/2}\right)$$

pentru a putea considera transformarea "cirip" ca o convoluţie.)

32.3. Implementări eficiente ale TFR

Deoarece aplicaţiile practice ale TFD, cum ar fi prelucrarea semnalelor, necesită viteză maximă, această secţiune examinează două implementări eficiente ale ale TFR. Mai întâi vom examina o versiune iterativă a algoritmului TFR care se execută în timp $\Theta(n \lg n)$, dar are o constantă mai mică ascunsă în notaţia Θ decât implementarea recursivă din secţiunea 32.2. Astfel, în scopul proiectării unui circuit paralel eficient pentru TFR, vom utiliza analiza care ne conduce la implementarea iterativă.

O implementare iterativă a TFR

Observăm întâi că ciclul **p**entru din liniile 10–13 ale procedurii TFR-RECURSIVĂ calculează valoarea $\omega_n^k y_k^{[1]}$ de două ori. În terminologia din domeniul compilatoarelor, o astfel de valoare este cunoscută sub numele de ***subexpresie comună***. Vom schimba ciclul **pentru** astfel încât să calculeze $\omega_n^k y_k^{[1]}$ o singură dată, memorând-o într-o variabilă temporară t.

pentru $k \leftarrow 0, n/2 - 1$ **execută**
  $t \leftarrow \omega y_k^{[1]}$
  $y_k \leftarrow y_k^{[0]} + t$
  $y_{k+(n/2)} \leftarrow y_k^{[0]} - t$
  $\omega \leftarrow \omega\omega_n$

Operaţia din acest ciclu, înmulţirea lui ω (care este egală cu ω_n^k) cu $y_k^{[1]}$, memorarea produsului în t, adunarea şi scăderea lui t din $y_k^{[0]}$, este cunoscută sub numele de ***operaţie fluture*** (butterfly operation) şi este prezentată schematic în figura 32.3. Vom arăta cum putem

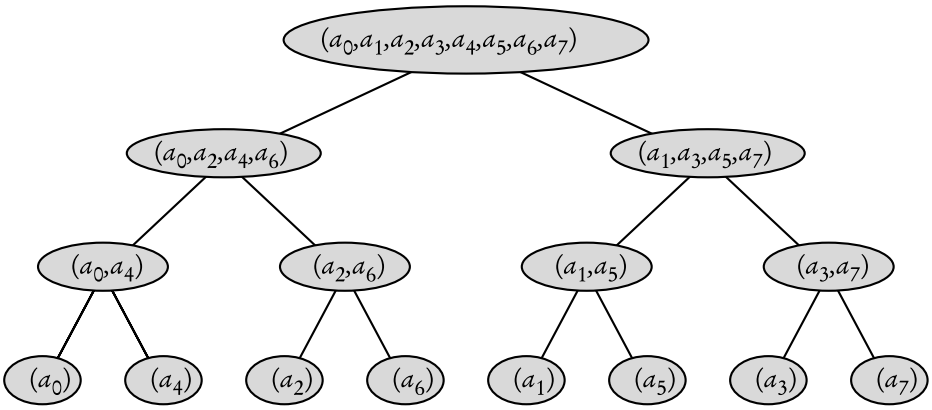


Figura 32.4 Cei trei vectori de intrare ai apelurilor recursive ale procedurii TFR-RECURSIVĂ. Apelul iniţial este pentru $n = 8$.

transforma un algoritm iterativ într-unul recursiv. În figura 32.4, am aranjat vectorii de intrare pentru apelurile recursive ale procedurii TFR-RECURSIVĂ într-o structură de arbore, unde apelul iniţial este pentru $n = 8$. Arborele are un nod pentru fiecare apel de procedură, etichetat cu vectorul de intrare corespunzător. Fiecare apel al procedurii TFR-RECURSIVĂ face două apeluri recursive, în afara cazului când primeşte un vector cu un element. Vom transforma primul apel fiul stâng şi al doilea fiul drept.

Examinând arborele, observăm că, dacă putem aranja elementele vectorului iniţial a în ordinea în care ele apar în frunze, putem imita execuţia procedurii TFR-RECURSIVĂ. Întâi, luăm elementele în perechi, calculăm TFD a fiecărei perechi, utilizând o operaţie fluture şi înlocuim perechea cu transformata sa Fourier discretă. Vectorul păstrează, atunci, $n/2$ TFD a câte două elemente. Apoi, luăm aceste $n/2$ perechi de TFD în perechi şi calculăm TFD a vectorilor cu 4 elemente care provin din execuţia a două operaţii fluture, înlocuind două TFD cu două elemente cu o TFD cu 4 elemente. Vectorul va păstra, apoi, $n/4$ TFD cu 4 elemente. Continuăm în acest mod până când vectorul păstrează două TFD a $n/2$ elemente care pot fi combinate în TFD finală cu n elemente.

Pentru a transforma această observaţie în cod, vom utiliza un tablou $A[0..n-1]$ care, iniţial, păstrează elementele vectorului de intrare a în ordinea în care apar în frunzele arborelui din figura 32.4. (Vom arăta, mai târziu, cum se poate determina această ordine.) Deoarece combinarea trebuie să se facă la fiecare nivel al arborelui, introducem o variabilă s, pentru a contoriza nivelurile, mergând de la 1 (în partea de jos, când combinăm perechile pentru a forma TFD a două elemente) la $\lg n$ (în vârf, când combinăm două TFD a $n/2$ elemente pentru a produce rezultatul final). De aceea algoritmul are următoarea structură:

1: **pentru** $s \leftarrow 1, \lg n$ **execută**
2: **pentru** $k \leftarrow 0, n-1, 2^s$ **execută**
3: combină două TFD a 2^{s-1} elemente din $A[k..k+2^{s-1}-1]$ şi $A[k+2^{s-1}..k+2^s-1]$ în TFD a 2^s elemente în $A[k..k+2^s-1]$

Putem exprima corpul ciclului (linia 3) printr-un pseudocod mai exact. Vom copia ciclul **pentru** din procedura TFR-RECURSIVĂ, identificând $y^{[0]}$ cu $A[k..k+2^{s-1}-1]$ şi $y^{[1]}$ cu $A[k+$

$2^{s-1}..k + 2^s - 1]$. Valoarea lui ω, utilizată în fiecare operaţie fluture, depinde de valoarea lui s; utilizăm ω_m, unde $m = 2^s$. (Introducem variabila m doar în scop de lizibilitate). Introducem o altă variabilă temporară u care ne permite să realizăm operaţia fluture pe loc (*in place*). Când înlocuim linia 3 a structurii generale prin corpul ciclului, obţinem pseudocodul următor, care formează baza algoritmului nostru final de TFR precum şi a implementării paralele pe care o vom prezenta mai târziu.

BAZA-TFR(a)
1: $n \leftarrow lungime[a]$ ▷ n este o putere a lui 2
2: **pentru** $s \leftarrow 1, \lg n$ **execută**
3: $m \leftarrow 2^s$
4: $\omega_m \leftarrow e^{2\pi i/m}$
5: **pentru** $k \leftarrow 0, n-1, m$ **execută**
6: $\omega \leftarrow 1$
7: **pentru** $j \leftarrow 0, m/2 - 1$ **execută**
8: $t \leftarrow \omega A[k + j + m/2]$
9: $u \leftarrow A[k + j]$
10: $A[k + j] \leftarrow u + t$
11: $A[k + j + m/2] \leftarrow u - t$
12: $\omega \leftarrow \omega\omega_m$

Prezentăm versiunea finală a codului iterativ pentru TFR, care inverseaz ă cele două cicluri interioare pentru a elimina unele calcule de index şi utilizează procedura auxiliară COPIERE-INVERSĂ-BIŢI(a, A) pentru a copia vectorul a în tabloul A în ordinea iniţială în care avem nevoie de valori.

TFR-ITERATIVĂ(a)
1: COPIERE-INVERSĂ-BIŢI(a, A)
2: $n \leftarrow lungime[a]$ ▷ n este o putere a lui 2
3: **pentru** $s \leftarrow 1, n$ **execută**
4: $m \leftarrow 2^s$
5: $\omega_m \leftarrow e^{2\pi i/m}$
6: $\omega \leftarrow 1$
7: **pentru** $j \leftarrow 0, m/2 - 1$ **execută**
8: **pentru** $k \leftarrow j, n-1, m$ **execută**
9: $t \leftarrow \omega A[k + j + m/2]$
10: $u \leftarrow A[k]$
11: $A[k] \leftarrow u + t$
12: $A[k + m/2] \leftarrow u - t$
13: $\omega \leftarrow \omega\omega_m$
14: **returnează** A

Cum mută COPIERE-INVERSĂ-BIŢI elementele din vectorul de intrare a în ordinea dorită în tabloul A? Ordinea în care frunzele apar în figura 32.4 este "ordinea binară inversă a biţilor". Adică, dacă rev(k) este întregul de lungime $\lg n$ format prin considerarea în ordine inversă a biţilor din reprezentarea binară a lui k, atunci vom pune elementul a_k în locul lui A[rev(k)].

De exemplu, în figura 32.4, frunzele apar în ordinea 0, 4, 2, 6, 1, 5, 3, 7; această secvenţă are reprezentările binare 000, 100, 010, 110, 001, 101, 011, 111, iar cu biţii inversaţi obţinem secvenţa 000, 001, 010, 011, 100, 101, 110, 111. Pentru a argumenta că, într-adevăr, dorim, în general, ordinea inversă a biţilor, să observăm că, la nivelul de sus al arborelui, indicii al căror bit de ordinul cel mai mic este zero sunt plasaţi în subarborele stâng, iar cei al căror bit mai puţin semnificativ este 1 sunt plasaţi în cel drept. Înlăturând biţii de ordinul cel mai mic la fiecare nivel, continuăm acest proces în jos până când obţinem ordinea binară inversă în frunze. Deoarece funcţia rev(k) se calculează uşor, procedura COPIERE-INVERSĂ-BIŢI poate fi scrisă după cum urmează:

COPIERE-INVERSĂ-BIŢI(a, A)
1: $n \leftarrow lungime[a]$
2: **pentru** $k = 0, n-1$ **execută**
3: $\quad A[\text{rev}(k)] \leftarrow a_k$

Implementarea iterativă a TFR se execută într-un timp $\Theta(n \lg n)$. Apelul procedurii COPIERE-INVERSĂ-BIŢI(a, A) se execută sigur în timp $O(n \lg n)$, deoarece iterăm de n ori şi putem oglindi (inversa) biţii unui întreg între 0 şi $n-1$ cu $\lg n$ biţi în timp $O(\lg n)$. (În practică, cunoaştem valoarea iniţială a lui n dinainte, aşa că putem construi o tabelă care să pună în corespondenţă k cu rev(k), şi astfel COPIERE-INVERSĂ-BIŢI se execută în timp $\Theta(n)$ cu o constantă ascunsă (*i.e.* conţinută în notaţia asimptotică) mică. Ca alternativă, putem utiliza o schemă amortizată inteligentă de inversare a unui contor binar, descrisă în problema 18-1. Pentru a încheia demonstraţia că TFR-ITERATIVĂ se execută în timp $\Theta(n \lg n)$, vom arăta că numărul $L(n)$ de execuţii ale corpului ciclului celui mai interior (liniile 9–12), este $\Theta(n \lg n)$. Avem:

$$L(n) = \sum_{s=1}^{\lg n} \sum_{j=0}^{2^{s-1}-1} \frac{n}{2^s} = \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} = \sum_{s=1}^{\lg n} \frac{n}{2} = \Theta(n \lg n).$$

Un circuit paralel pentru TFR

Putem exploata multe dintre proprietăţile care ne permit să implementăm un algoritm TFR iterativ eficient pentru a produce un algoritm paralel eficient pentru TFR. (Pentru descrierea unui model de circuit combinaţional, vezi capitolul 29.) Circuitul combinaţional TFR-PARALELĂ care calculează TFR a n intrări, pentru $n = 8$, apare în figura 32.5. Circuitul începe cu o permutare prin inversarea biţilor intrării, urmată de $\lg n$ stadii, fiecare constând din $n/2$ operaţii fluture executate în paralel. Adâncimea circuitului este $\Theta(\lg n)$.

Partea cea mai din stânga a circuitului TFR-PARALELĂ realizează permutarea prin inversarea biţilor, iar restul imită procedura iterativă BAZA-TFR. Vom profita de avantajul că fiecare iteraţie a celui mai exterior ciclu **p**entru (liniile 3–12) realizează $n/2$ operaţii fluture independente care pot fi executate în paralel. Valoarea lui s din fiecare iteraţie din interiorul BAZA-TFR corespunde stadiului operaţiilor fluture ilustrate în figura 32.5. În interiorul stadiului s, pentru $s = 1, 2, \ldots, \lg n$, există $n/2^s$ grupuri de operaţii fluture (corespunzând fiecărei valori a lui k din BAZA-TFR) cu 2^{s-1} fluturi pe grup (corespunzând fiecărei valori a lui j din BAZA-TFR). Operaţiile fluture din figura 32.5 corespund operaţiilor fluture din ciclul cel mai interior (liniile 8–11 ale procedurii BAZA-TFR). De notat, de asemenea, că valorile lui ω utilizate în operaţii fluture corespund celor din BAZA-TFR; în stadiul s utilizăm $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, unde $m = 2^s$.

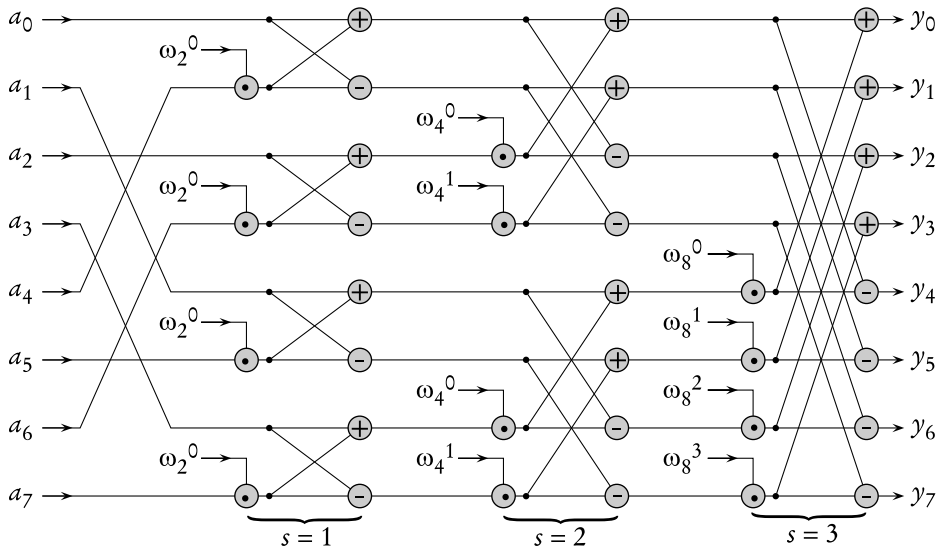


Figura 32.5 Un circuit combinaţional TFR-PARALELĂ care calculează TFR, ilustrat pentru $n = 8$ intrări. Stadiile operaţiilor fluture sunt etichetate pentru a corespunde ciclului celui mai exterior al procedurii BAZA-TFR. O TFR pentru n intrări poate fi calculată cu adâncimea $\Theta(\lg n)$, folosind $\Theta(n \lg n)$ elemente combinaţionale.

Exerciţii

32.3-1 Arătaţi cum calculează TFR-ITERATIVĂ TFD a vectorului de intrare (0,2,3,-1,4,5,7,9).

32.3-2 Arătaţi cum se implementează un algoritm TFR în care permutarea prin inversarea biţilor să apară la sfârşitul calculului, în loc să apară la început. (*Indicaţie:* consideraţi TFD inversă).

32.3-3 Câte elemente de adunare, scădere şi înmulţire sunt necesare în circuitul TFR-PARALELĂ descris în această secţiune, pentru a calcula TFD_n (presupunem că este nevoie numai de un fir pentru a duce un număr dintr-un loc în altul)?

32.3-4 $\star$ Presupunem că sumatorii dintr-un circuit TFR uneori greşesc într-un mod care produce, întotdeauna, o ieşire 0 indiferent de intrare. Presupunem că greşeşte exact un sumator, dar că nu ştim care anume. Descrieţi modul în care se poate identifica sumatorul defect, furnizând intrări circuitului TFR global şi observând ieşirile. Încercaţi să realizacti o procedură eficientă.

Probleme

32-1 Înmulţire divide şi stăpâneşte

a. Arătaţi cum se pot înmulţi două polinoame de grad I $ax + b$ şi $cx + d$, utilizând numai trei înmulţiri. (*Indicaţie:* una dintre înmulţiri este $(a + b)(c + d)$.)

b. Daţi doi algoritmi divide şi stăpâneşte pentru înmulţirea a două polinoame cu limita de grad n care se execută în timp $\Theta(n^{\lg 3})$. Primul algoritm ar putea împărţi coeficienţii polinoamelor de intrare într-o jumătate stângă şi una dreaptă, iar cel de-al doilea după cum indicele lor este par sau impar.

c. Arătaţi că doi întregi de n biţi pot fi înmulţiţi în $O(n^{\lg 3})$ paşi, unde fiecare pas operează pe cel mult un număr constant de valori de un bit.

32-2 Matrice Toeplitz

O ***matrice Toeplitz*** este o matrice de dimensiune $n \times n$, $A = (a_{ij})$, astfel încât $a_{ij} = a_{i-1,j-1}$ pentru $i = 2, 3, \ldots, n$ şi $j = 2, 3, \ldots, n$.

a. Este suma a două matrice Toeplitz, în mod necesar, o matrice Toeplitz? Ce se poate spune despre produs?

b. Descrieţi un mod de reprezentare a matricelor Toeplitz, astfel ca două matrice Toeplitz $n \times n$ să poată fi adunate în timp $O(n)$.

c. Daţi un algoritm în timp $O(n \lg n)$ de înmulţire a unei matrice Toeplitz cu un vector de lungime n. Utilizaţi reprezentarea din partea (b).

d. Daţi un algoritm eficient de înmulţire a două matrice Toeplitz de dimensiune $n \times n$. Analizaţi timpul său de execuţie.

32-3 Evaluarea tuturor derivatelor unui polinom într-un punct

Fie un polinom $A(x)$ cu limita de grad n şi derivata sa de ordinul t este definită prin

$$A^{(t)}(x) = \begin{cases} A(x) & \text{dacă} \quad t = 0 \\ \frac{d}{dx} A^{(t-1)}(x) & \text{dacă} \quad 1 \le t \le n-1 \\ 0 & \text{dacă} \quad t \ge n. \end{cases}$$

Dându-se reprezentarea prin coeficienţi $(a_0, a_1, \ldots, a_{n-1})$ a lui $A(x)$ şi un punct x_0, dorim să determinăm $A^{(t)}(x_0)$ pentru $t = 0, 1, \ldots, n-1$.

a. Fie coeficienţii $b_0, b_1, \ldots, b_{n-1}$, astfel încât

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j.$$

Arătaţi cum se calculează $A^{(t)}(x_0)$, pentru $t = 0, 1, \ldots, n-1$, în timp $O(n)$.

b. Explicaţi cum putem găsi $b_0, b_1, \ldots, b_{n-1}$ în timp $O(n \lg n)$, dându-se $A(x_0 + \omega_n^k)$, pentru $k = 0, 1, \ldots, n-1$.

c. Demonstraţi că

$$A(x_0 + \omega_n^k) = \sum_{i=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right),$$

unde $f(j) = a_j \cdot j!$ şi

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{dacă} \quad -(n-1) \le l \le 0 \\ 0 & \text{dacă} \quad 1 \le l \le n-1. \end{cases}$$

d. Explicaţi cum se poate evalua $A(x_0 + \omega_n^k)$ pentru $k = 0, 1, \ldots, n-1$ în timp $O(n \lg n)$.

32-4 Evaluarea polinoamelor în puncte multiple

Am văzut că problema evaluării unui polinom cu limita de grad $n-1$ poate fi rezolvată în timp $O(n)$, utilizând schema lui Horner. Am descoperit, de asemenea, că un astfel de polinom poate fi evaluat în toate cele n rădăcini complexe de ordinul n ale unităţii în timp $O(n \lg n)$ utilizând TFR. Vom vedea cum putem să evaluăm un polinom cu limita de grad n în n puncte arbitrare în timp $O(n \lg^2 n)$.

Pentru aceasta, vom utiliza faptul că putem calcula polinomul rest al împărţirii unui polinom la altul în timp $O(n \lg n)$, rezultat pe care îl vom accepta fără demonstraţie. De exemplu, restul împărţirii lui $3x^3+x^2-3x+1$ la x^2+x+2 este $(3x^3+x^2-3x+1) \bmod (x^2+x+2) = -7x+5$. Dându-se reprezentarea prin coeficienţi a unui polinom $A(x) = \sum_{k=0}^{n-1} a_k x^k$ şi n puncte $x_0, x_1, \ldots, x_{n-1}$, dorim să calculăm n valori $A(x_0), A(x_1), \ldots, A(x_{n-1})$. Pentru $0 \le i \le j \le n-1$, definim polinoamele $P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)$ şi $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$. De notat că $Q_{ij}(x)$ are limita de grad cel mult $j - i$.

a. Demonstraţi că $A(x) \bmod (x - z) = A(z)$ pentru orice punct z.

b. Demonstraţi că $Q_{kk}(x) = A(x_k)$ şi că $Q_{0,n-1}(x) = A(x)$.

c. Demonstraţi că, pentru $i \le k \le j$, avem $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ şi $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

d. Daţi un algoritm în timp $O(n \lg^2 n)$ pentru a evalua $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

32-5 TFR utilizând aritmetică modulară

Aşa cum s-a definit, transformata Fourier discretă necesită utilizarea numerelor complexe, ceea ce poate conduce la o pierdere de precizie datorată erorilor de rotunjire. Pentru anumite probleme, se ştie că răspunsul conţine doar numere întregi şi este de dorit să utilizăm o variantă a TFR bazată pe aritmetica modulară pentru a garanta că răspunsul este calculat exact. Un exemplu de astfel de problemă este aceea a înmulţirii a două polinoame cu coeficienţi numere întregi. Exerciţiul 32.2-6 dă o astfel de abordare ce utilizează un modul de lungime $\Omega(n)$ biţi pentru a realiza o TFD pe n puncte. Această problemă oferă o altă abordare care utilizează un modul de lungime rezonabilă $O(\lg n)$; ea necesită înţelegerea materialului din capitolul 33. Fie n o putere a lui 2.

a. Să presupunem că dorim să căutăm cel mai mic k, astfel ca $p = kn + 1$ să fie prim. Vom da un argument euristic simplu, care justifică de ce ne aşteptăm ca valoarea lui k să fie aproximativ $\lg n$. (Valoarea lui k ar putea fi mult mai mare sau mult mai mică, dar este rezonabil să examinăm candidatul $O(\lg n)$ ca valoare a lui k în medie.) Cum ne aşteptăm să fie lungimea lui p comparativ cu lungimea lui n?

Fie g un generator al lui $\mathbb{Z}_p^*$ şi fie $w = g^k \bmod p$.

b. Argumentaţi că TFD şi inversa TFD sunt operaţii bine definite modulo p, unde w este utilizată ca rădăcină principală de ordinul n a unităţii.

c. Argumentaţi că TFR şi inversa sa pot fi făcute să lucreze modulo p în timp $O(n \lg n)$, unde operaţiile pe cuvinte de lungime $O(\lg n)$ durează o unitate de timp. Presupunem că algoritmul primeşte la intrare p şi w.

d. Calculaţi TFD modulo $p = 17$ a vectorului (0, 5, 3, 7, 7, 2, 1, 6). De notat că $g = 3$ este un generator al lui $\mathbb{Z}_{17}^*$.

Note bibliografice

Press, Flannery, Teukolsky şi Vetterling [161, 162] conţine o bună descriere a transformatei Fourier rapide şi a aplicaţiilor sale. Pentru o excelentă introducere în prelucrarea semnalelor, o aplicaţie binecunoscută şi răspândită a domeniului TFR, recomandăm textul lui Oppenheim şi Willsky [153].

Mulţi autori atribuie lui Cooley şi Tuckey [51] descoperirea TFR prin anii '60. De fapt, TFR a fost descoperită cu mult timp înainte, dar importanţa sa nu a fost pe deplin înţeleasă înaintea apariţiei calculatoarelor numerice moderne. Press, Flannery, Teukolsky şi Vetterling atribuie originile metodei lui Runge şi König (1924).

33 Algoritmi din teoria numerelor

Teoria numerelor a fost cândva considerată ca fiind un subiect frumos, dar fără a avea o mare valoare în matematica pură. Astăzi, algoritmii din teoria numerelor se utilizează des, datorită, în mare parte, inventării schemelor criptografice bazate pe numere prime mari. Flexibilitatea acestor scheme rezidă în abilitatea noastră de a găsi uşor numere prime mari, în timp ce securitatea lor rezidă în imposibilitatea noastră de a factoriza produsul de numere prime mari. Acest capitol prezintă unele aspecte ale teoriei numerelor şi algoritmii asociaţi care stau la baza unor astfel de aplicaţii.

În secţiunea 33.1 se introduc conceptele de bază din teoria numerelor, cum ar fi divizibilitatea, echivalenţa modulară şi factorizarea unică. În secţiunea 33.2 se studiază unul dintre cei mai vechi algoritmi: algoritmul lui Euclid pentru calculul celui mai mare divizor comun a doi întregi. În secţiunea 33.3 se revăd conceptele aritmeticii modulare. În secţiunea 33.4 se studiază mulţimea multiplilor unui număr dat a, modulo n şi se arată cum se găsesc toate soluţiile ecuaţiei $ax \equiv b \pmod{n}$ utilizând algoritmul lui Euclid. Teorema chinezească a restului este prezentată în secţiunea 33.5. În secţiunea 33.6 se consideră puterile modulo n ale unui număr dat a şi se prezintă un algoritm de ridicare repetată la pătrat pentru calculul eficient al lui $a^b \bmod n$, când a, b şi n sunt date. Această operaţie se află la baza testului eficient al numerelor prime. Secţiunea 33.7 descrie sistemul de criptare RSA cu cheie publică. În secţiunea 33.8 se descrie o randomizare a testului de număr prim care poate fi utilizată pentru a găsi eficient numere prime mari, ceea ce este o sarcină esenţială în crearea cheilor pentru sistemul de criptare RSA. În final, secţiunea 33.9 trece în revistă o metodă euristică simplă, dar eficientă pentru factorizarea întregilor mici. Este neobişnuit faptul că factorizarea este o problemă pe care unii oameni ar dori-o de nerezolvat, deoarece securitatea RSA depinde de dificultăţile de factorizare ale întregilor mari.

Dimensiunea intrărilor şi costul calculelor aritmetice

Deoarece vom lucra cu numere întregi mari, avem nevoie să ne formăm anumite idei privind dimensiunea unei intrări şi costul operaţiilor aritmetice elementare.

În acest capitol, prin "intrare mare", de obicei, se înţelege o intrare care conţine "întregi mari" şi nu o intrare care conţine "mulţi întregi" (ca în cazul sortării). Astfel, vom măsura dimensiunea unei intrări în termenii *numărului de biţi* necesari pentru a reprezenta acea intrare şi nu doar în numărul de întregi de la intrare. Un algoritm având ca date de intrare numerele întregi $a_1, a_2, \ldots, a_k$ este un ***algoritm polinomial*** dacă se execută într-un timp polinomial în $\lg a_1, \lg a_2, \ldots, \lg a_k$, adică în timp polinomial în lungimile datelor de intrare codificate binar.

În majoritatea situaţiilor din această carte, am considerat că este convenabil să tratăm operaţiile aritmetice elementare (înmulţiri, împărţiri sau calculul resturilor) ca operaţii primitive care necesită un timp de o unitate. Calculând numărul de operaţii aritmetice în acest fel, avem o bază pentru a face o estimare rezonabilă a timpului real de execuţie al algoritmului pe calculator. Operaţiile elementare pot fi consumatoare de timp, mai ales când intrările sunt mari. Devine convenabil să măsurăm cât de multe ***operaţii pe biţi*** necesită un algoritm de teoria numerelor. În acest mod, o înmulţire a două numere întregi reprezentate pe β biţi, prin metode obişnuite, utilizează $\Theta(\beta^2)$ operaţii pe biţi. Similar, operaţia de împărţire a unui întreg reprezentat pe β biţi printr-un întreg mai scurt sau operaţia de calcul a restului împărţirii unui întreg reprezentat

pe β biţi la un întreg mai scurt, poate fi făcută într-un timp $\Theta(\beta^2)$ prin algoritmi simpli (vezi exerciţiul 33.1-11). Sunt cunoscute metode mai rapide. De exemplu, o metodă simplă divide şi stăpâneşte pentru înmulţirea a doi întregi reprezentaţi pe β biţi are un timp de execuţie de $\Theta(\beta^{\lg_2 3})$, iar cea mai rapidă metodă cunoscută are un timp de execuţie de $\Theta(\beta \lg \beta \lg\lg \beta)$. Din motive practice, totuşi algoritmul cu timpul $\Theta(\beta^2)$ este cel mai bun şi vom folosi această limită ca o bază pentru analiza noastră.

În acest capitol, algoritmii sunt, în general, analizaţi atât în termenii numărului de operaţii aritmetice cât şi ai numărului de operaţii pe biţi pe care le necesită.

33.1. Noţiuni elementare de teoria numerelor

Această secţiune permite o scurtă trecere în revistă a noţiunilor din teoria elementară a numerelor privind mulţimea numerelor întregi $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ şi mulţimea numerelor naturale $\mathbb{N} = \{0, 1, 2, \ldots\}$.

Divizibilitate şi divizori

Noţiunea ca un întreg să fie divizibil cu un altul este de bază în teoria numerelor. Notaţia $d \mid a$ (se citeşte "d ***divide pe*** a") înseamnă că $a = kd$ pentru un anumit întreg k. Orice întreg divide pe 0. Dacă $a > 0$ şi $d \mid a$, atunci $|d| \leq |a|$. Dacă $d|a$, mai spunem că a este ***multiplu*** al lui d. Dacă d nu divide pe a, vom scrie $d \nmid a$.

Dacă $d \mid a$ şi $d \geq 0$, spunem că d este un ***divizor*** al lui a. Să observăm că $d \mid a$ dacă şi numai dacă $-d \mid a$, aşa că nu se pierde din generalitate dacă definim divizorii ca fiind nenegativi, înţelegând că negativul oricărui divizor al lui a este, de asemenea, divizor al lui a. Un divizor al unui întreg a este cel puţin 1 şi nu este mai mare dacât $|a|$. De exemplu, divizorii lui 24 sunt 1, 2, 3, 4, 6, 8, 12 şi 24.

Orice întreg a este divizibil prin ***divizorii triviali*** 1 şi a. Divizorii proprii ai lui a se numesc ***factori***. De exemplu, factorii lui 20 sunt 2, 4, 5 şi 10.

Numere prime şi compuse

Un întreg $a > 1$ ai cărui divizori sunt numai divizorii triviali 1 şi a se spune că este un număr ***prim*** (sau mai simplu un ***prim***). Numerele prime au multe proprietăţi speciale şi joacă un rol critic în teoria numerelor. Numerele prime mici sunt, în ordine:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, \ldots.$$

Exerciţiul 33.1-1 cere să se demonstreze că există o infinitate de numere prime. Un întreg $a > 1$ care nu este prim este un ***număr compus*** (sau mai simplu ***compus***). De exemplu, 39 este compus deoarece $3 \mid 39$. Se spune că întregul 1 este o ***unitate*** şi nu este nici prim şi nici compus. Similar, întregul 0 şi toţi întregii negativi nu sunt nici primi şi nici compuşi.

Teorema împărţirii, resturi şi echivalenţă modulară

Dacă se dă un întreg n, întregii pot fi partiţionaţi în întregi care sunt multipli de n şi cei care nu sunt multipli de n. Teoria numerelor se bazează pe o rafinare a acestei partiţionări obţinută

prin clasificarea nemultiplilor lui n potrivit resturilor lor când sunt împărţiţi la n. Această rafinare se bazează pe următoarea teoremă. Demonstraţia ei nu va fi dată aici (vezi, de exemplu, Niven şi Zuckerman [151]).

Teorema 33.1 (Teorema împărţirii) Pentru orice întreg a şi orice întreg pozitiv n, există q şi r unici, astfel încât $0 \leq r < n$ şi $a = qn + r$. ∎

Valoarea $q = \lfloor a/n \rfloor$ este ***câtul*** împărţirii. Valoarea $r = a \bmod n$ este ***restul*** (sau ***reziduul***) împărţirii. $n \mid a$ dacă şi numai dacă $a \bmod n = 0$. Rezultă că

$$a = \lfloor a/n \rfloor n + (a \bmod n) \tag{33.1}$$

sau

$$a \bmod n = a - \lfloor a/n \rfloor n. \tag{33.2}$$

Având noţiunea de rest bine definită, este convenabil să furnizăm o notaţie specială pentru a indica egalitatea resturilor. Dacă $(a \bmod n) = (b \bmod n)$, vom scrie $a \equiv b \pmod{n}$ şi vom spune că a este ***echivalent*** cu b, modulo n. Cu alte cuvinte, $a \equiv b \pmod{n}$ dacă a şi b au acelaşi rest la împărţirea cu n. Echivalent, $a \equiv b \pmod{n}$ dacă şi numai dacă $n \mid (b - a)$. Scriem $a \not\equiv b \pmod{n}$ dacă a nu este echivalent cu b modulo n. De exemplu, $61 \equiv 6 \pmod{11}$. De asemenea, $-13 \equiv 22 \equiv 2 \pmod{5}$.

Întregii pot fi împărţiţi în n clase de echivalenţă potrivit resturilor lor modulo n. ***Clasa de echivalenţă modulo n*** care conţine un întreg a este

$$[a]_n = \{a + kn : k \in \mathbb{Z}\}.$$

De exemplu, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$; alte notaţii pentru această mulţime sunt $[-4]_7$ şi $[10]_7$. Scrierea $a \in [b]_n$ reprezintă acelaşi lucru cu scrierea $a \equiv b \pmod{n}$. Mulţimea tuturor acestor clase de echivalenţă este

$$\mathbb{Z}_n = \{[a]_n : 0 \leq a \leq n - 1\}. \tag{33.3}$$

Adesea se foloseşte definiţia

$$\mathbb{Z}_n = \{0, 1, \ldots, n - 1\}, \tag{33.4}$$

care ar trebui citită ca fiind echivalentă cu relaţia (33.3), subînţelegând că 0 reprezintă $[0]_n$, 1 reprezintă $[1]_n$ şi aşa mai departe; fiecare clasă este reprezentată prin cel mai mic element nenegativ. Fundamentul claselor de echivalenţă trebuie, totuşi reţinut. De exemplu, o referire la -1 ca membru al lui $\mathbb{Z}_n$ este o referire la $[n - 1]_n$ deoarece $-1 \equiv n - 1 \pmod{n}$.

Divizori comuni şi cel mai mare divizor comun

Dacă d este un divizor al lui a şi al lui b, atunci d este un ***divizor comun*** al lui a şi b. De exemplu, divizorii lui 30 sunt 1, 2, 3, 5, 6, 10, 15 şi 30, divizorii comuni ai lui 24 şi 30 sunt 1, 2, 3 şi 6. Să observăm că 1 este un divizor comun pentru orice doi întregi.

O proprietate importantă a divizorilor comuni este

$$d \mid a \text{ şi } d \mid b \text{ implică } d \mid (a + b) \text{ şi } d \mid (a - b). \tag{33.5}$$

Mai general, avem

$$d \mid a \text{ şi } d \mid b \text{ implică } d \mid (ax + by) \tag{33.6}$$

pentru orice întregi x şi y. Dacă $a \mid b$, atunci sau $|a| \leq |b|$ sau $b = 0$, ceea ce implică faptul că

$$a \mid b \text{ şi } b \mid a \text{ implică } a = \pm b. \tag{33.7}$$

Cel mai mare divizor comun a doi întregi a şi b, nu ambii nuli, este cel mai mare dintre divizorii comuni ai lui a şi b; se notează cu $\text{cmmdc}(a, b)$. De exemplu, $\text{cmmdc}(24, 30) = 6$, $\text{cmmdc}(5, 7) = 1$ şi $\text{cmmdc}(0, 9) = 9$. Dacă a şi b nu sunt ambii nuli, atunci $\text{cmmdc}(a, b)$ este un întreg între 1 şi $\min(|a|, |b|)$. Prin definiţie, $\text{cmmdc}(0, 0) = 0$. Această definiţie este necesară pentru a putea standardiza proprietăţile funcţiei cmmdc (cum ar fi relaţia (33.11) de mai jos).

Proprietăţile elementare ale funcţiei cmmdc sunt:

$$\text{cmmdc}(a, b) = \text{cmmdc}(b, a), \tag{33.8}$$

$$\text{cmmdc}(a, b) = \text{cmmdc}(-a, b), \tag{33.9}$$

$$\text{cmmdc}(a, b) = \text{cmmdc}(|a|, |b|), \tag{33.10}$$

$$\text{cmmdc}(a, 0) = |a|, \tag{33.11}$$

$$\text{cmmdc}(a, ka) = |a| \text{ pentru orice } k \in \mathbb{Z}. \tag{33.12}$$

Teorema 33.2 Dacă a şi b sunt întregi, nu ambii nuli, atunci $\text{cmmdc}(a, b)$ este cel mai mic element pozitiv al mulţimii $\{ax + by : x, y \in \mathbb{Z}\}$ de combinaţii liniare ale lui a şi b.

Demonstraţie. Fie s o cea mai mică (pozitivă) combinaţie liniară între a şi b, astfel încât $s = ax + by$ pentru $x, y \in \mathbb{Z}$. Fie $q = \lfloor a/s \rfloor$. Relaţia 33.2 implică

$$a \bmod s = a - qs = a - q(ax + by) = a(1 - qx) + b(-qy).$$

astfel, $a \bmod s$ este o combinaţie liniară a lui a şi b. Dar, întrucât $a \bmod s < s$, avem $a \bmod s = 0$, deoarece s este cea mai mică astfel de combinaţie liniară pozitivă. De aceea, $s \mid a$ şi, din motive analoge, $s \mid b$. Astfel, s este un divizor comun al lui a şi b şi $\text{cmmdc}(a, b) \geq s$. Ecuaţia (33.6) implică faptul că $\text{cmmdc}(a, b) \mid s$ deoarece $\text{cmmdc}(a, b)$ divide atât pe a cât şi pe b şi s este o combinaţie liniară a lui a şi b. Dar $\text{cmmdc}(a, b) \mid s$ şi $s > 0$ implică faptul că $\text{cmmdc}(a, b) \leq s$. Combinând $\text{cmmdc}(a, b) \geq s$ şi $\text{cmmdc}(a, b) \leq s$, obţinem $\text{cmmdc}(a, b) = s$; concluzionăm că s este cel mai mare divizor comun al lui a şi b. ∎

Corolarul 33.3 Pentru orice întregi a şi b, dacă $d \mid a$ şi $d \mid b$, atunci $d \mid \text{cmmdc}(a, b)$.

Demonstraţie. Acest corolar rezultă din (33.6), deoarece $\text{cmmdc}(a, b)$ este o combinaţie liniară a lui a şi b conform teoremei 33.2. ∎

Corolarul 33.4 Pentru orice întregi a şi b şi orice întreg n nenegativ,

$$\text{cmmdc}(an, bn) = n \, \text{cmmdc}(a, b).$$

Demonstraţie. Dacă $n = 0$, corolarul este trivial. Dacă $n > 0$, atunci $\text{cmmdc}(an, bn)$ este cel mai mic element pozitiv al mulţimii $\{anx + bny\}$, care este de n ori cel mai mic element pozitiv al mulţimii $\{ax + by\}$. ∎

Corolarul 33.5 Pentru orice numere întregi pozitive n, a şi b, dacă $n \mid ab$ şi $\text{cmmdc}(a, n) = 1$, atunci $n \mid b$.

Demonstraţie. Demonstraţia este lăsată pe seama cititorului (exerciţiul 33.1-4). ∎

Întregi relativ primi

Două numere întregi a, b sunt ***relativ prime*** dacă singurul lor divizor comun este 1, adică, dacă $\text{cmmdc}(a, b) = 1$. De exemplu, 8 şi 15 sunt relativ prime, deoarece divizorii lui 8 sunt 1, 2, 4 şi 8, în timp ce divizorii lui 15 sunt 1, 3, 5 şi 15. Teorema următoare afirmă că, dacă două numere întregi sunt fiecare relativ prime cu un întreg p, atunci produsul lor este relativ prim cu p.

Teorema 33.6 Pentru orice numere întregi a, b şi p, dacă $\text{cmmdc}(a, p) = 1$ şi $\text{cmmdc}(b, p) = 1$, atunci $\text{cmmdc}(ab, p) = 1$.

Demonstraţie. Din teorema 33.2 rezultă că există înumerele întregi x, y, x' şi y', astfel încât

$$ax + py = 1,$$
$$bx' + py' = 1.$$

Înmulţind aceste ecuaţii şi rearanjându-le, avem

$$ab(xx') + p(ybx' + y'ax + pyy') = 1.$$

Întrucât 1 este o combinaţie liniară a lui ab şi p, un apel la teorema 33.2 completează demonstraţia. ∎

Spunem că înumerele întregi $n_1, n_2, \ldots, n_k$ sunt ***relativ prime două câte două*** dacă, ori de câte ori $i \neq j$, avem $\text{cmmdc}(n_i, n_j) = 1$.

Unicitatea factorizării

Un fapt elementar, dar important, despre divizibilitatea cu numere prime este următoarea teoremă.

Teorema 33.7 Pentru toate numerele prime p şi toţi întregii a, b, dacă $p \mid ab$, atunci $p \mid a$ sau $p \mid b$.

Demonstraţie. Presupunem prin absurd că $p \mid ab$ dar $p \nmid a$ şi $p \nmid b$. Atunci, $\text{cmmdc}(a, p) = 1$ şi $\text{cmmdc}(b, p) = 1$, deoarece singurii divizori ai lui p sunt 1 şi p, iar prin ipoteză p nu divide nici pe a nici pe b. Teorema 33.6 implică faptul că $\text{cmmdc}(ab, p) = 1$, contrazicând ipoteza că $p \mid ab$, deoarece $p \mid ab$ implică $\text{cmmdc}(ab, p) = p$. Această contradicţie încheie demonstraţia. ∎

O consecinţă a teoremei 33.7 este faptul că un număr întreg are o factorizare unică în numere prime.

Teorema 33.8 (Unicitatea factorizării) Un întreg compus a poate fi scris exact într-un singur mod ca produs de forma

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$

unde p_i sunt prime, $p_1 < p_2 < \cdots < p_r$, iar e_i sunt numere întregi pozitive.

Demonstraţie. Demonstraţia este lăsată pe seama cititorului (exerciţiul 33.1-10). ∎

Ca un exemplu, numărul 6000 poate fi factorizat în mod unic astfel $2^4 \cdot 3 \cdot 5^3$.

Exerciţii

33.1-1 Demonstraţi că există o infinitate de numere prime. (*Indicaţie:* se va arăta că nici unul din numerele prime $p_1, p_2, \ldots, p_k$ nu divide $(p_1 p_2 \cdots p_k) + 1$.)

33.1-2 Demonstraţi că, dacă $a \mid b$ şi $b \mid c$, atunci $a \mid c$.

33.1-3 Demonstraţi că, dacă p este prim şi $0 < k < p$, atunci $\text{cmmdc}(k, p) = 1$.

33.1-4 Demonstraţi corolarul 33.5.

33.1-5 Demonstraţi că, dacă p este prim şi $0 < k < p$, atunci $p \mid \binom{p}{k}$. Să se concluzioneze că, pentru orice numere întregi a, b şi orice număr prim p,

$$(a + b)^p \equiv a^p + b^p (\text{mod } p).$$

33.1-6 Demonstraţi că, oricare ar fi a şi b numere întregi astfel încât $a \mid b$ şi $b > 0$, atunci

$$(x \bmod b) \bmod a = x \bmod a$$

pentru orice x. Demonstraţi, în aceleaşi ipoteze, că

$$x \equiv y \ (\text{mod } b) \text{ implică } x \equiv y \ (\text{mod } a)$$

pentru orice numere întregi x şi y.

33.1-7 Pentru orice număr întreg $k > 0$, spunem că n este o ***k putere*** dacă există un număr întreg a, astfel încât $a^k = n$. Spunem că $n > 1$ este o ***putere nevidă*** dacă el este o k putere pentru un anumit număr întreg $k > 1$. Arătaţi cum se determină, într-un timp polinomial în β, dacă un număr întreg dat n reprezentat pe β biţi este o putere nevidă.

33.1-8 Demonstraţi relaţiile (33.8)–(33.12).

33.1-9 Arătaţi că operatorul cmmdc este asociativ. Adică, trebuie să arătaţi că, pentru orice numere întregi a, b şi c,

$$\text{cmmdc}(a, \text{cmmdc}(b, c)) = \text{cmmdc}(\text{cmmdc}(a, b), c).$$

33.1-10 ⋆ Demonstraţi teorema 33.8.

33.1-11 Daţi algoritmi eficienţi pentru împărţirea unui număr întreg reprezentat pe β biţi la un număr întreg mai mic şi pentru determinarea restului împărţirii unui număr întreg reprezentat pe β biţi la un număr întreg mai mic. Algoritmii trebuie să se execute într-un timp de ordinul $O(\beta^2)$.

33.1-12 Daţi un algoritm eficient de conversie a unui număr întreg (binar) reprezentat pe β biţi în reprezentare zecimală. Argumentaţi faptul că, dacă înmulţirea sau împărţirea numerelor întregilor a căror lungime este cel mult β necesită un timp $M(\beta)$, atunci conversia din binar în zecimal poate fi realizată într-un timp de ordinul $\Theta(M(\beta) \lg \beta)$. (*Indicaţie:* să se utilizeze o concepţie divide şi stăpâneşte, care obţine jumătăţile superioară şi inferioară ale rezultatului cu recurenţe separate.)

33.2. Cel mai mare divizor comun

În această secţiune, utilizăm algoritmul lui Euclid pentru a calcula eficient cel mai mare divizor comun a doi întregi. Analiza timpului de execuţie ne conduce la o legătură surprinzătoare cu numerele Fibonacci, care ne oferă date de intrare defavorabile pentru algoritmul lui Euclid.

În această secţiune ne restrângem la numere întregi nenegative. Această restricţie este justificată prin relaţia (33.10), care afirmă că $\text{cmmdc}(a, b) = \text{cmmdc}(|a|, |b|)$.

În principiu, putem calcula $\text{cmmdc}(a, b)$ pentru întregii pozitivi a şi b din factorizarea în numere prime a lui a şi b. Într-adevăr, dacă

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \tag{33.13}$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r}, \tag{33.14}$$

folosind exponenţi 0 pentru a obţine aceleaşi numere prime $p_1, p_2, \ldots, p_r$ atât pentru a cât şi pentru b, atunci

$$\text{cmmdc}(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)} \tag{33.15}$$

Aşa cum se va arăta în secţiunea 33.9, cel mai bun algoritm pentru factorizare, până în prezent, nu se execută într-un timp polinomial. Astfel, această concepţie de calcul al celui mai mare divizor comun este puţin probabil să producă un algoritm eficient.

Algoritmul lui Euclid pentru calculul celui mai mare divizor comun se bazează pe următoarea teoremă.

Teorema 33.9 (Teorema CMMDC recursiv) Pentru orice întreg nenegativ a şi pentru orice întreg pozitiv b,

$$\text{cmmdc}(a, b) = \text{cmmdc}(b, a \bmod b).$$

Demonstraţie. Vom arăta că $\text{cmmdc}(a, b)$ şi $\text{cmmdc}(b, a \bmod b)$ se divid unul pe altul, aşa că, prin relaţia (33.7), ei trebuie să fie egali (deoarece sunt nenegativi).

Arătăm mai întâi că $\text{cmmdc}(a, b) \mid \text{cmmdc}(b, a \bmod b)$. Dacă $d = \text{cmmdc}(a, b)$, atunci $d \mid a$ şi $d \mid b$. Din (33.2), $(a \bmod b) = a - qb$, unde $q = \lfloor a/b \rfloor$. Întrucât $(a \bmod b)$ este o combinaţie liniară a lui a şi b, relaţia (33.6) implică faptul că $d \mid (a \bmod b)$. De aceea, întrucât $d \mid b$ şi $a \mid (a \bmod b)$, corolarul 33.3 implică faptul că $d \mid \text{cmmdc}(b, a \bmod b)$ sau echivalent,

$$\text{cmmdc}(a, b) \mid \text{cmmdc}(b, a \bmod b). \tag{33.16}$$

A arăta că $\text{cmmdc}(b, a \bmod b) \mid \text{cmmdc}(a, b)$ este aproape acelaşi lucru. Dacă acum $d = \text{cmmdc}(b, a \bmod b)$, atunci $d \mid b$ şi $d \mid (a \bmod b)$. Întrucât $a = qb + (a \bmod b)$, unde $q = \lfloor a/b \rfloor$, deci a este o combinaţie liniară dintre b şi $(a \bmod b)$. Din ecuaţia (33.6), deducem că $d \mid a$. Întrucât $d \mid b$ şi $d \mid a$, avem $d \mid \text{cmmdc}(a, b)$ conform corolarului 33.3 sau echivalentul,

$$\text{cmmdc}(b, a \bmod b) \mid \text{cmmdc}(a, b). \tag{33.17}$$

Pentru a completa demonstraţia, folosim relaţia (33.7) pentru a combina relaţiile (33.16) şi (33.17). ∎

Algoritmul lui Euclid

Următorul algoritm cmmdc este descris în *Elementele* lui Euclid (circa 300 î.Hr.), deşi el poate avea o origine mai timpurie. Este scris ca un program recursiv bazat direct pe teorema 33.9. Intrările a şi b sunt întregi nenegativi arbitrari.

```
EUCLID(a, b)
1: dacă b = 0 atunci
2:    returnează a
3: altfel
4:    returnează EUCLID(b, a mod b)
```

Ca un exemplu de execuţie al algoritmului EUCLID, considerăm calculul lui cmmdc$(30, 21)$:

$$\text{EUCLID}(30, 21) = \text{EUCLID}(21, 9) = \text{EUCLID}(9, 3) = \text{EUCLID}(3, 0) = 3.$$

În acest calcul există trei apeluri recursive ale lui EUCLID.

Corectitudinea lui EUCLID este garantată de teorema 33.9 şi de faptul că, dacă algoritmul returnează pe a în linia 2, atunci $b = 0$, deci cmmdc(a, b) = cmmdc$(a, 0) = a$, conform relaţiei (33.11). Algoritmul nu poate intra în ciclu infinit, deoarece al doilea argument descreşte strict la fiecare apel recursiv. De aceea, EUCLID se termină totdeauna cu răspunsul corect.

Timpul de execuţie al algoritmului lui Euclid

Analizăm timpul de execuţie pentru EUCLID în cazul cel mai defavorabil, în funcţie de dimensiunile lui a şi b. Presupunem, cu o mică pierdere a generalităţii, că $a > b \geq 0$. Această presupunere poate fi justificată prin observaţia că, dacă $b > a \geq 0$, atunci EUCLID(a, b) face imediat apelul la EUCLID(b, a). Adică, dacă primul argument este mai mic decât cel de al doilea, atunci EUCLID realizează un apel recursiv pentru a-şi permuta parametrii şi, apoi, continuă. Similar, dacă $b = a > 0$, procedura se termină după un apel recursiv deoarece $a \bmod b = 0$.

Timpul total de execuţie al lui EUCLID este proporţional cu numărul apelurilor recursive pe care le face. Analiza noastră face apel la numerele F_k ale lui Fibonacci, definite prin recurenţa (2.13).

Lema 33.10 Dacă $a > b \geq 0$ şi apelul EUCLID(a, b) execută $k \geq 1$ apeluri recursive, atunci $a \geq F_{k+2}$ şi $b \geq F_{k+1}$.

Demonstraţie. Demonstraţia se face prin inducţie în raport cu k. Fie $k = 1$, atunci, $b \geq 1 = F_2$ şi, deoarece $a > b$, trebuie să avem $a \geq 2 = F_3$. Deoarece $b > (a \bmod b)$, la fiecare apel recursiv, primul argument este strict mai mare decât al doilea, de aceea, presupunerea că $a > b$ se adevereşte pentru fiecare apel recursiv.

Presupunem acum că lema este adevărată când se realizează $k - 1$ apeluri recursive; vom demonstra atunci că ea este adevărată şi pentru k apeluri recursive. Întrucât $k > 0$, avem $b > 0$, iar EUCLID(a, b) apelează recursiv pe EUCLID$(b, a \bmod b)$ care, la rândul său, execută $k - 1$ apeluri recursive. Ipotezele inducţiei implică faptul că $b \geq F_{k+1}$ (astfel se demonstrează o parte a lemei), iar $(a \bmod b) \geq F_k$. Avem

$$b + (a \bmod b) = b + (a - \lfloor a/b \rfloor b) \leq a,$$

deoarece $a > b > 0$ implică $\lfloor a/b \rfloor \geq 1$. Astfel,

$$a \geq b + (a \bmod b) \geq F_{k+1} + F_k = F_{k+2}.$$

■

Următoarea teoremă este un corolar imediat al acestei leme.

Teorema 33.11 (Teorema lui Lamé) Pentru orice întreg $k \geq 1$, dacă $a > b \geq 0$ şi $b < F_{k+1}$, atunci apelul procedurii EUCLID(a, b) execută mai puţin de k apeluri recursive. ■

Putem arăta că limita superioară din teorema 33.11 este cea mai bună posibilă. În consecinţă, numerele lui Fibonacci consecutive sunt cele mai defavorabile cazuri de intrare pentru EUCLID. Întrucât EUCLID(F_3, F_2) execută exact un apel recursiv şi deoarece pentru $k \geq 2$ avem $F_{k+1} \bmod F_k = F_{k-1}$, avem:

$$\text{cmmdc}(F_{k+1}, F_k) = \text{cmmdc}(F_k, (F_{k+1} \bmod F_k)) = \text{cmmdc}(F_k, F_{k-1}).$$

Astfel, EUCLID(F_{k+1}, F_k) iterează *exact* de $k - 1$ ori, atingând limita superioară din teorema 33.11.

Deoarece F_k este aproximativ $\phi^k/\sqrt{5}$, unde ϕ este raportul de aur $(1 + \sqrt{5})/2$ definit prin ecuaţia (2.14), numărul apelurilor recursive în EUCLID este $O(\lg b)$. (Vezi exerciţiul 33.2-5 pentru o limită mai strânsă.) Dacă se aplică EUCLID la două numere reprezentate pe β biţi, atunci procedura va efectua $O(\beta)$ operaţii aritmetice şi $O(\beta^3)$ operaţii pe biţi (presupunând că înmulţirile şi împărţirile numerelor reprezentate pe β biţi necesită $O(\beta^2)$ operaţii pe biţi). Problema 33-2 cere să arătăm că $O(\beta^2)$ este o limită a numărului de operaţii pe biţi.

Forma extinsă a algoritmului lui Euclid

Rescriem algoritmul lui Euclid pentru a determina informaţii suplimentare utile. În special, extindem algoritmul pentru a calcula coeficienţii întregi x şi y, astfel încât

$$d = \text{cmmdc}(a, b) = ax + by. \tag{33.18}$$

Să observăm că x şi y pot fi negativi sau zero. Mai târziu, aceşti coeficienţi vor fi utili în calculul inverselor multiplicative modulare.

Procedura EUCLID-EXTINS are la intrare o pereche arbitrară de întregi nenegativi şi returnează un triplet de forma (d, x, y) care satisface relaţia (33.18).

EUCLID-EXTINS(a, b)
1: **dacă** $b = 0$ **atunci**
2: **returnează** $(a, 1, 0)$
3: $(d', x', y') \leftarrow$ EUCLID-EXTINS$(b, a \bmod b)$
4: $(d, x, y) \leftarrow (d', y', x' - \lfloor a/b \rfloor y')$
5: **returnează** (d, x, y)

Figura 33.1 ilustrează execuţia lui EUCLID-EXTINS pentru calculul lui cmmdc$(99, 78)$.

Procedura EUCLID-EXTINS este o variantă a procedurii EUCLID. Linia 1 este echivalentă cu testul "$b = 0$" din linia 1 a procedurii EUCLID. Dacă $b = 0$, EUCLID-EXTINS returnează nu

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	—	3	1	0

Figura 33.1 Un exemplu de aplicare a lui EUCLID-EXTINS asupra intrărilor 99 şi 78. Fiecare linie arată un nivel al recursivităţii: valorile de intrare a şi b, valoarea calculată $\lfloor a/b \rfloor$ şi valorile returnate d, x şi y. Tripletul returnat (d, x, y) devine tripletul (d', x', y') utilizat în calcul la nivelul mai înalt, următor al recursivităţii. Apelul EUCLID-EXTINS$(99, 78)$ returnează $(3, -11, 14)$, astfel încât cmmdc$(99, 78) = 3$ şi cmmdc$(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

numai $d = a$ în linia 2, ci şi coeficienţii $x = 1$ şi $y = 0$, astfel încât $a = ax + by$. Dacă $b \neq 0$, EUCLID-EXTINS calculează întâi (d', x', y'), astfel încât $d' =$ cmmdc$(b, a \bmod b)$ şi

$$d' = bx' + (a \bmod b)y'. \tag{33.19}$$

Ca la EUCLID, avem în acest caz $d =$ cmmdc$(a, b) =$ cmmdc$(b, a \bmod b) = d'$. Pentru a obţine pe x şi y, astfel încât $d = ax + by$, începem prin rescrierea ecuaţiei (33.19) utilizând relaţia $d = d'$ şi relaţia (33.2):

$$d = bx' + (a - \lfloor a/b \rfloor b)y' = ay' + b(x' - \lfloor a/b \rfloor y').$$

Astfel, alegând $x = y'$ şi $y = x' - \lfloor a/b \rfloor y'$, se verifică relaţia $d = ax + by$, ceea ce demonstrează corectitudinea lui EUCLID-EXTINS.

Întrucât numărul apelurilor recursive din EUCLID este egal cu numărul apelurilor recursive din EUCLID-EXTINS, timpul de execuţie pentru cele două proceduri este acelaşi, abstracţie făcând de un factor constant. Adică, pentru $a > b > 0$, numărul apelurilor recursive este $O(\lg b)$.

Exerciţii

33.2-1 Demonstraţi că relaţiile (33.13)–(33.14) implică relaţia (33.15).

33.2-2 Calculaţi valorile (d, x, y) care sunt ieşirile la apelul EUCLID-EXTINS$(899, 493)$.

33.2-3 Demonstraţi că, oricare ar fi întregii a, k şi n,

cmmdc$(a, n) =$ cmmdc$(a + kn, n)$.

33.2-4 Rescrieţi procedura EUCLID într-o formă iterativă care să folosească numai o cantitate constantă de memorie (adică, memorează numai un număr constant de valori întregi).

33.2-5 Dacă $a > b \geq 0$, arătaţi că apelul lui EUCLID(a, b) implică cel mult $1 + \log_\phi b$ apeluri recursive. Să se îmbunătăţească această limită la $1 + \log_\phi(b/\text{cmmdc}(a, b))$.

33.2-6 Ce returnează EUCLID-EXTINS(F_{k+1}, F_k)? Demonstraţi corectitudinea răspunsului dat.

33.2-7 Verificaţi ieşirea (d, x, y) a lui EUCLID-EXTINS(a, b) arătând că, dacă $d \mid a, d \mid b$ şi $d = ax + by$, atunci $d = \text{cmmdc}(a, b)$.

33.2-8 Definiţi funcţia cmmdc pentru mai mult de două argumente prin relaţia recursivă $\text{cmmdc}(a_0, a_1, \ldots, a_n) = \text{cmmdc}(a_0, \text{cmmdc}(a_1, \ldots, a_n))$. Arătaţi că cmmdc returnează acelaşi răspuns, independent de ordinea în care se specifică argumentele sale. Arătaţi cum se găsesc $x_0, x_1, \ldots, x_n$, astfel încât $\text{cmmdc}(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \cdots + a_nx_n$. Arătaţi că numărul de împărţiri realizat de algoritmul respectiv este de ordinul $O(n + \lg(\max_i a_i))$.

33.2-9 Definiţi $\text{cmmmc}(a_1, a_2, \ldots, a_n)$ ca fiind ***cel mai mic multiplu comun*** al numerelor întregi $a_1, a_2, \ldots, a_n$, adică cel mai mic întreg nenegativ care este multiplu al fiecărui a_i. Arătaţi cum se calculează eficient $\text{cmmmc}(a_1, a_2, \ldots, a_n)$ folosind operaţia *cmmdc* (cu două argumente) ca subrutină.

33.2-10 Demonstraţi că n_1, n_2, n_3 şi n_4 sunt relativ prime două câte două, dacă şi numai dacă $\text{cmmdc}(n_1n_2, n_3n_4) = \text{cmmdc}(n_1n_3, n_2n_4) = 1$. Arătaţi că, în cazul general, $n_1, n_2, \ldots, n_k$ sunt relativ prime două câte două dacă şi numai dacă o mulţime de $\lceil \lg k \rceil$ perechi de numere derivate din n_i sunt relativ prime.

33.3. Aritmetică modulară

Informativ, putem să ne gândim la aritmetica modulară ca la aritmetica obişnuită pentru numere întregi, exceptând faptul că, dacă lucrăm modulo n, atunci fiecare rezultat x este înlocuit printr-un element din $\{0, 1, \ldots, n-1\}$, adică este echivalent cu x modulo n (adică, x este înlocuit cu $x \bmod n$). Acest model informal este suficient dacă ne limităm la operaţiile de adunare, scădere şi înmulţire. Un model mai formal pentru aritmetica modulară, pe care îl dăm acum, se descrie mai bine în cadrul teoriei grafurilor.

Grupuri finite

Un ***grup*** $(S, \oplus)$ este o mulţime S, împreună cu o operaţie binară $\oplus$ definită pe S, pentru care se îndeplinesc următoarele proprietăţi.

1. ***Închidere***: Oricare ar fi $a, b \in S$, avem $a \oplus b \in S$.
2. ***Element neutru***: Există un element $e \in S$, astfel încât $e \oplus a = a \oplus e = a$ pentru orice $a \in S$.
3. ***Asociativitate***: Oricare ar fi $a, b, c \in S$, avem $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
4. ***Invers***: Pentru orice $a \in S$, există un element unic $b \in S$, astfel încât $a \oplus b = b \oplus a = e$.

De exemplu, considerăm grupul familiar $(\mathbb{Z}, +)$ al întregilor $\mathbb{Z}$ cu operaţia de adunare: 0 este unitatea, iar inversul lui a este $-a$. Dacă într-un grup $(S, \oplus)$ legea este ***comutativă***, $a \oplus b = b \oplus a$ oricare ar fi $a, b \in S$, atunci el este un ***grup abelian***. Dacă un grup $(S, \oplus)$ satisface condiţia $|S| < \infty$, atunci este un ***grup finit***.

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

Figura 33.2 Două grupuri finite. Clasele de echivalenţă sunt notate prin elementele lor reprezentative. (**a**) Grupul $(\mathbb{Z}_6, +_6)$. (**b**) Grupul $(\mathbb{Z}_{15}^*, \cdot_{15})$.

Grupuri definite prin adunare şi înmulţire modulară

Putem forma două grupuri abeliene finite folosind adunarea şi înmulţirea modulo n, unde n este un întreg pozitiv. Aceste grupuri se bazează pe clasele de echivalenţă a întregilor modulo n, definite în secţiunea 33.1.

Pentru a defini un grup pe $\mathbb{Z}_n$, este nevoie de operaţii binare corespunzătoare, pe care le obţinem redefinind operaţiile obişnuite de adunare şi înmulţire. Este uşor să definim aceste operaţii în $\mathbb{Z}_n$ deoarece clasele de echivalenţă a doi întregi determină în mod unic clasa de echivalenţă a sumei şi a produsului lor. Adică, dacă $a \equiv a' (\bmod n)$ şi $b \equiv b' (\bmod n)$, atunci

$$\begin{aligned} a + b &\equiv a' + b' (\bmod n) \\ ab &\equiv a'b' (\bmod n). \end{aligned}$$

Astfel, definim adunarea şi înmulţirea modulo n, notate $+_n$ şi $\cdot_n$, după cum urmează:

$$[a]_n +_n [b]_n = [a + b]_n, [a]_n \cdot_n [b]_n = [ab]_n.$$

(Scăderea poate fi definită similar pe $\mathbb{Z}_n$ prin $[a]_n -_n [b]_n = [a - n]_n$, dar împărţirea este mai complicată, aşa cum vom vedea.) Aceste fapte justifică practica obişnuită şi convenabilă de a utiliza cel mai mic element nenegativ al fiecărei clase de echivalenţă ca reprezentant al ei când se fac calcule în $\mathbb{Z}_n$. Adunarea, scăderea şi înmulţirea se efectuează obişnuit asupra reprezentanţilor, dar fiecare rezultat x se înlocuieşte cu reprezentantul clasei sale (adică, prin $x \bmod n$).

Utilizând această definiţie a adunării modulo n, definim ***grupul aditiv modulo n*** prin $(\mathbb{Z}_n, +_n)$. Dimensiunea grupului aditiv modulo n este $|\mathbb{Z}_n| = n$. În figura 33.2 (**a**) avem tabela operaţiei $+_6$ pentru grupul $(\mathbb{Z}_6, +_6)$.

Teorema 33.12 Sistemul $(\mathbb{Z}_n, +_n)$ este un grup abelian finit.

Demonstraţie. Asociativitatea şi comutativitatea lui $+_n$ rezultă din asociativitatea şi comutativitatea lui $+$:

$$([a]_n +_n [b]_n) +_n [c]_n = [(a+b)+c]_n = [a+(b+c)]_n = [a]_n +_n ([b]_n +_n [c]_n),$$

$$[a]_n +_n [b]_n = [a+b]_n = [b+a]_n = [b]_n +_n [a]_n.$$

Elementul unitate al lui $(\mathbb{Z}_n, +_n)$ este 0 (adică $[0]_n$). Inversul (aditiv) al unui element a (adică, $[a]_n$) este elementul $-a$ (adică, $[-a]_n$ sau $[n-a]_n$), deoarece $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$. ∎

Utilizând definiţia înmulţirii modulo n, definim ***grupul multiplicativ modulo n*** prin $(\mathbb{Z}_n^\star, \cdot_n)$. Elementele acestui grup aparţin mulţimii $\mathbb{Z}_n^\star$ a elementelor din $\mathbb{Z}_n$ care sunt relativ prime cu n:

$$\mathbb{Z}_n^\star = \{[a]_n \in \mathbb{Z}_n : \text{ cmmdc}(a,n) = 1\}.$$

Pentru a vedea că $\mathbb{Z}_n^\star$ este bine definit, observăm că, pentru $0 \le a < n$, avem $a \equiv (a + kn) \pmod n$ pentru orice întregi k. Din exerciţiul 33.2-3, rezultă că $\text{cmmdc}(a,n) = 1$ implică $\text{cmmdc}(a+kn, n) = 1$ pentru orice întregi k. Deoarece $[a]_n = \{a + kn : k \in \mathbb{Z}\}$, mulţimea $\mathbb{Z}_n^\star$ este bine definită. Un exemplu de astfel de grup este

$$\mathbb{Z}_{15}^\star = \{1, 2, 4, 7, 8, 11, 13, 14\},$$

unde operaţia grupului este înmulţirea modulo 15. (Aici notăm un element $[a]_{15}$ cu a.) În figura 33.2 (**b**) avem grupul $(\mathbb{Z}_{15}^\star, \cdot_{15})$. De exemplu, $8 \cdot 11 = 13 \pmod{15}$, în $\mathbb{Z}_{15}^\star$. Unitatea pentru acest grup este 1.

Teorema 33.13 Sistemul $(\mathbb{Z}_n^\star, \cdot_n)$ este un grup abelian finit.

Demonstraţie. Teorema 33.6 implică faptul că $(\mathbb{Z}_n^\star, \cdot_n)$ este închis. Asociativitatea şi comutativitatea pot fi demonstrate pentru $\cdot_n$, aşa cum au fost ele demonstrate pentru $+_n$, în demonstraţia teoremei 33.12. Elementul unitate este $[1]_n$. Pentru a arăta existenţa inversului, fie a un element al lui $\mathbb{Z}_n^\star$ şi fie (d, x, y) ieşirea lui EUCLID-EXTINS(a, n). Atunci, $d = 1$, deoarece $a \in \mathbb{Z}_n^\star$ şi

$$ax + ny = 1$$

sau, echivalentul,

$$ax \equiv 1 \pmod n.$$

Astfel, $[x]_n$ este un invers multiplicativ al lui $[a]_n$, modulo n. Demonstraţia că inversul este unic definit o amânăm până la corolarul 33.26. ∎

Când vom lucra cu grupurile $(\mathbb{Z}_n, +_n)$ şi $(\mathbb{Z}_n^\star, \cdot_n)$ în restul acestui capitol, vom folosi convenţia practică de a nota clasele de echivalenţă prin elementele lor reprezentative şi vom nota operaţiile $+_n$ şi $\cdot_n$ prin notaţiile obişnuite $+$ respectiv $\cdot$ (sau juxtapunere). De asemenea, echivalenţele modulo n pot fi interpretate ca relaţii în $\mathbb{Z}_n$. De exemplu, următoarele două afirmaţii sunt echivalente:

$$\begin{aligned} ax &\equiv b \pmod n, \\ [a]_n \cdot_n [x]_n &= [b]_n. \end{aligned}$$

Altă convenţie ar fi următoarea: când operaţia e subînţeleasă din context ne vom referi la grupul $(S, \oplus)$ doar prin S. Astfel, ne putem referi la grupurile $(\mathbb{Z}_n, +_n)$ şi $(\mathbb{Z}_n^\star, \cdot_n)$ prin $\mathbb{Z}_n$ şi, respectiv, $\mathbb{Z}_n^\star$.

Inversul (multiplicativ) al unui element a se notează cu $(a^{-1} \bmod n)$. Împărţirea în $\mathbb{Z}_n^\star$ se defineşte prin ecuaţia $a/b \equiv ab^{-1} (\bmod n)$. De exemplu, în $\mathbb{Z}_{15}^\star$ avem $7^{-1} \equiv 13 \pmod{15}$, deoarece $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, aşa că $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

Dimensiunea lui $\mathbb{Z}_n^\star$ se notează cu $\phi(n)$. Această funcţie, cunoscută ca ***funcţia phi a lui Euler***, satisface ecuaţia

$$\phi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right), \tag{33.20}$$

unde p variază peste toate numerele prime care-l divid pe n (inclusiv n, dacă n este prim). Nu vom demonstra această formulă (aici). Intuitiv, începem cu o listă de n resturi $\{0, 1, \cdots, n-1\}$ şi apoi, pentru fiecare număr prim p care divide pe n, eliminăm de pe listă fiecare multiplu al lui p. De exemplu, deoarece divizorii primi ai lui 45 sunt 3 şi 5,

$$\phi(45) = 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) = 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) = 24.$$

Dacă p este prim, atunci $\mathbb{Z}_p^\star = \{1, 2, \cdots, p-1\}$, iar

$$\phi(p) = p - 1. \tag{33.21}$$

Dacă n este compus, atunci $\phi(n) < n - 1$.

Subgrupuri

Dacă $(S, \oplus)$ este un grup, $S' \subseteq S$ şi $(S', \oplus)$ este, de asemenea, un grup, atunci se spune că $(S', \oplus)$ este un ***subgrup*** al lui $(S, \oplus)$. De exemplu, mulţimea numerelor întregi pare formează un subgrup al întregilor faţă de operaţia de adunare. Următoarea teoremă furnizează un instrument util pentru recunoaşterea subgrupurilor.

Teorema 33.14 (O submulţime închisă nevidă a unui grup finit este un subgrup) Dacă $(S, \oplus)$ este un grup finit şi S' este orice submulţime a lui S astfel încât $a \oplus b \in S'$ pentru orice $a, b \in S'$, atunci $(S', \oplus)$ este un subgrup al lui $(S, \oplus)$.

Demonstraţie. Lăsăm demonstraţia pe seama cititorului (exerciţiul 33.3-2). ∎

De exemplu, mulţimea $\{0, 2, 4, 6\}$ formează un subgrup al lui $\mathbb{Z}_8$, deoarece este închisă în raport cu operaţia $+$ (adică este închisă faţă de $+_8$).

Următoarea teoremă furnizează o restricţie extrem de importantă asupra dimensiunii unui subgrup.

Teorema 33.15 (Teorema lui Lagrange) Dacă $(S, \oplus)$ este un grup finit şi $(S', \oplus)$ este un subgrup al lui $(S, \oplus)$, atunci $|S'|$ este un divizor al lui $|S|$. ∎

Despre un subgrup S' al unui grup S se spune că este subgrup ***propriu*** dacă $S' \neq S$.

Următorul corolar va fi util în analiza procedurii de test a numerelor prime a lui Miller-Rabin din secţiunea 33.8.

Corolarul 33.16 Dacă S' este un subgrup propriu al unui grup finit S, atunci $|S'| \leq |S|/2$. ∎

Subgrupuri generate de un element

Teorema 33.14 furnizează un mod interesant de a produce un subgrup al unui grup finit $(S, \oplus)$: se alege un element a şi se iau toate elementele care pot fi generate din a utilizând operaţia grupului. Mai exact, definim $a^{(k)}$ pentru $k \geq 1$ prin:

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus a \cdots \oplus a}_{k}.$$

De exemplu, pentru $a = 2$ în grupul $\mathbb{Z}_6$, secvenţa $a^{(1)}, a^{(2)}, \ldots$ este 2, 4, 0, 2, 4, 0, 2, 4, 0, ...

În grupul $\mathbb{Z}_n$, avem $a^{(k)} = ka \bmod n$, iar în grupul $\mathbb{Z}_n^\star$, avem $a^{(k)} = a^k \bmod n$. ***Subgrupul generat de*** a, notat $\langle a \rangle$ sau $(\langle a \rangle, \oplus)$, este definit prin:

$$\langle a \rangle = \{a^{(k)} : k \geq 1\}.$$

Spunem că a ***generează*** subgrupul $\langle a \rangle$ sau că a este ***un generator*** al lui $\langle a \rangle$. Deoarece S este finit, $\langle a \rangle$ este o submulţime finită a lui S, posibil să coincidă cu S. Întrucât asociativitatea lui $\oplus$ implică

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)},$$

$\langle a \rangle$ este închisă şi, deci, prin teorema 33.14, $\langle a \rangle$ este un subgrup al lui S. De exemplu, în $\mathbb{Z}_6$ avem:

$$\begin{aligned} \langle 0 \rangle &= \{0\}, \\ \langle 1 \rangle &= \{0, 1, 2, 3, 4, 5\}, \\ \langle 2 \rangle &= \{0, 2, 4\}. \end{aligned}$$

Similar, în $\mathbb{Z}_7^\star$, avem:

$$\begin{aligned} \langle 1 \rangle &= \{1\}, \\ \langle 2 \rangle &= \{1, 2, 4\}, \\ \langle 3 \rangle &= \{1, 2, 3, 4, 5, 6\}. \end{aligned}$$

Ordinul lui a (în grupul S), notat $\mathrm{ord}(a)$, se defineşte prin cel mai mic $t > 0$ astfel încât $a^{(t)} = e$.

Teorema 33.17 Pentru orice grup finit $(S, \oplus)$ şi orice $a \in S$, ordinul unui element este egal cu dimensiunea subgrupului pe care-l generează, sau $\mathrm{ord}(a) = |\langle a \rangle|$.

Demonstraţie. Fie, $t = \mathrm{ord}(a)$. Deoarece $a^{(t)} = e$ şi $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ pentru $k \geq 1$, dacă $i > t$, atunci $a^{(i)} = a^{(j)}$ pentru un anumit $j < i$. Astfel, nu există elemente noi după $a^{(t)}$ şi $\langle a \rangle = \{a^{(1)}, a^{(2)}, \ldots, a^{(t)}\}$. Pentru a arăta că $|\langle a \rangle| = t$, presupunem prin absurd că $a^{(i)} = a^{(j)}$ pentru i, j care satisfac relaţia $1 \leq i < j \leq t$. Atunci, $a^{(i+k)} = a^{(j+k)}$ pentru $k \geq 0$. Dar aceasta implică faptul că $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, ceea ce este o contradicţie deoarece $i + (t - j) < t$, dar t este cea mai mică valoare pozitivă astfel încât $a^{(t)} = e$. De aceea, fiecare element al secvenţei $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ este distinct şi $|\langle a \rangle| = t$. ∎

Corolarul 33.18 Secvenţa $a^{(1)}, a^{(2)}, \ldots$ este periodică având perioada $t = \mathrm{ord}(a)$; adică, $a^{(i)} = a^{(j)}$ dacă şi numai dacă $i \equiv j \pmod{t}$. ∎

Conform corolarului de mai sus, are sens să definim pe $a^{(0)}$ ca fiind e şi $a^{(i)}$ prin $a^{(i \bmod t)}$ pentru orice număr întreg i.

Corolarul 33.19 Dacă $(S, \oplus)$ este un grup finit cu unitatea e, atunci

$$a^{(|S|)} = e,$$

pentru orice $a \in S$.

Demonstraţie. Teorema lui Lagrange implică faptul că $\text{ord}(a) \mid |S|$ şi astfel $S \equiv 0 \pmod{t}$, unde $t = \text{ord}(a)$. ∎

Exerciţii

33.3-1 Studiaţi tabela operaţiei de grup pentru grupurile $(\mathbb{Z}_4, +_4)$ şi $(\mathbb{Z}_5^\star, \cdot_5)$. Arătaţi că aceste grupuri sunt izomorfe, stabilind o corespondenţă biunivocă α între elementele lor, astfel încât $a + b \equiv c \pmod 4$ dacă şi numai dacă $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod 5$.

33.3-2 Demonstraţi teorema 33.14.

33.3-3 Arătaţi că, dacă p este prim şi e este un întreg pozitiv, atunci

$$\phi(p^e) = p^{e-1}(p-1).$$

33.3-4 Arătaţi că, pentru orice $n > 1$ şi pentru orice $a \in \mathbb{Z}_n^\star$, funcţia $f_a : \mathbb{Z}_n^\star \to \mathbb{Z}_n^\star$ definită prin $f_a(x) = ax \bmod n$ este o permutare a lui $\mathbb{Z}_n^\star$.

33.3-5 Scrieţi toate subgrupurile lui $\mathbb{Z}_9$ şi ale lui $\mathbb{Z}_{13}^\star$.

33.4. Rezolvarea ecuaţiilor liniare modulare

O problemă practică importantă este problema determinării soluţiilor ecuaţiei:

$$ax \equiv b (\bmod\ n), \tag{33.22}$$

unde $n > 0$. Presupunem că a, b şi n se dau, iar noi trebuie să-i găsim pe acei x modulo n, care satisfac ecuaţia (33.22). Pot exista zero, una sau mai multe astfel de soluţii.

Fie $\langle a \rangle$ subgrupul lui $\mathbb{Z}_n$ generat de a. Deoarece $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, ecuaţia (33.22) are o soluţie dacă şi numai dacă $b \in \langle a \rangle$. Teoarema lui Lagrange (teorema 33.15) ne spune că $|\langle a \rangle|$ trebuie să fie un divizor al lui n. Următoarea teoremă ne dă o caracterizare precisă a lui $\langle a \rangle$.

Teorema 33.20 Pentru orice numere întregi, pozitive a şi n, dacă $d = \text{cmmdc}(a, n)$, atunci:

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\}, \tag{33.23}$$

şi, astfel,

$$|\langle a \rangle| = n/d.$$

Demonstraţie. Începem prin a arăta că $d \in \langle a \rangle$. Să observăm că EUCLID-EXTINS(a, n) determină întregii x' şi y', astfel încât $ax' + ny' = d$. Avem $ax' \equiv d \pmod{n}$, astfel că $d \in \langle a \rangle$.

Întrucât $d \in \langle a \rangle$, înseamnă că orice multiplu al lui d aparţine lui $\langle a \rangle$, deoarece un multiplu al unui multiplu al lui a este tot un multiplu al lui a. Astfel, $\langle a \rangle$ conţine orice element din $\{0, d, 2d, \ldots, ((n/d) - 1)d\}$. Adică, $\langle d \rangle \subseteq \langle a \rangle$.

Arătăm acum că $\langle a \rangle \subseteq \langle d \rangle$. Dacă $m \in \langle a \rangle$, atunci $m = ax \bmod n$ pentru un anumit întreg x şi, astfel, $m = ax + ny$ pentru un anumit întreg y. Însă, $d \mid a$ şi $d \mid n$ şi, astfel, $d \mid m$ din relaţia (33.6). De aceea $m \in \langle d \rangle$.

Combinând aceste rezultate, avem $\langle a \rangle = \langle d \rangle$. Pentru a vedea că $|\langle a \rangle| = n/d$, să observăm că există exact n/d multipli ai lui d între 0 şi $n - 1$, inclusiv. ∎

Corolarul 33.21 Ecuaţia $ax \equiv b \pmod{n}$ este rezolvabilă în necunoscuta x dacă şi numai dacă $\text{cmmdc}(a, n) \mid b$. ∎

Corolarul 33.22 Ecuaţia $ax \equiv b \pmod{n}$ fie are d soluţii distincte modulo n, unde $d = \text{cmmdc}(a, n)$, fie nu are nici o soluţie.

Demonstraţie. Dacă $ax \equiv b \pmod{n}$ are o soluţie, atunci $b \in \langle a \rangle$. Şirul $ai \bmod n$, pentru $i = 0, 1, \ldots$ este periodic având perioada $|\langle a \rangle| = n/d$, conform corolarului 33.18. Dacă $b \in \langle a \rangle$, atunci b apare exact de d ori în şirul $ai \bmod n$, pentru $i = 0, 1, \ldots, n - 1$, deoarece blocul de lungime (n/d) al valorilor lui $\langle a \rangle$ se repetă exact de d ori când i creşte de la 0 la $n - 1$. Indicii x ai acestor d poziţii sunt soluţiile ecuaţiei $ax \equiv b \pmod{n}$. ∎

Teorema 33.23 Fie $d = \text{cmmdc}(a, n)$ şi presupunem că $d = ax' + ny'$ pentru anumiţi întregi x' şi y' (de exemplu, cei calculaţi prin EUCLID-EXTINS). Dacă $d \mid b$, atunci ecuaţia $ax \equiv b \pmod{n}$ are una din soluţiile sale valoarea x_0, unde

$$x_0 = x'(b/d) \bmod n.$$

Demonstraţie. Deoarece $ax' \equiv d \pmod{n}$, avem

$$\begin{aligned} ax_0 &\equiv ax'(b/d) && \pmod{n} \\ &\equiv d(b/d) && \pmod{n} \\ &\equiv b && \pmod{n}, \end{aligned}$$

deci, x_0 este o soluţie a lui $ax \equiv b \pmod{n}$. ∎

Teorema 33.24 Presupunem că ecuaţia $ax \equiv b \pmod{n}$ este rezolvabilă (adică $d \mid b$, unde $d = \text{cmmdc}(a, n)$) şi că x_0 este o soluţie oarecare a acestei ecuaţii. Atunci, această ecuaţie are exact d soluţii distincte modulo n, date de $x_i = x_0 + i(n/d)$ pentru $i = 1, 2, \ldots, d - 1$.

Demonstraţie. Deoarece $n/d > 0$ şi $0 \leq i(n/d) < n$ pentru $i = 0, 1, \ldots, d - 1$, valorile $x_0, x_1, \ldots, x_{d-1}$ sunt toate distincte modulo n. Din periodicitatea şirului $ai \bmod n$ (corolarul 33.18), rezultă că, dacă x_0 este o soluţie a lui $ax \equiv b \pmod{n}$, atunci orice x_i este o soluţie. Prin corolarul 33.22, există exact d soluţii, astfel că $x_0, x_1, \ldots, x_{d-1}$ trebuie să fie exact acestea. ∎

Până acum am dezvoltat matematica necesară pentru a rezolva ecuaţia $ax \equiv b \pmod{n}$; următorul algoritm afişează toate soluţiile acestei ecuaţii. Intrările a şi b sunt întregi arbitrari, iar n este un întreg pozitiv arbitrar.

REZOLVĂ-ECUAŢIE-LINIARĂ-MODULARĂ(a, b, n)
1: $(d, x', y') \leftarrow$EUCLID-EXTINS(a, n)
2: **dacă** $d \mid b$ **atunci**
3: $x_0 \leftarrow x'(b/d) \bmod n$
4: **pentru** $i \leftarrow 0$, $d - 1$ **execută**
5: scrie $(x_0 + i(n/d)) \bmod n$
6: **altfel**
7: scrie "Nu există soluţie"

Pentru a exemplifica funcţionarea acestei proceduri, considerăm ecuaţia $14x \equiv 30 \pmod{100}$ (aici $a = 14, b = 30$ şi $n = 100$). Apelând EUCLID-EXTINS în linia 1, obţinem $(d, x, y) = (2, -7, 1)$. Deoarece $2 \mid 30$, se execută liniile 3–5. În linia 3 calculăm $x_0 = (-7)(15) \bmod 100 = 95$. Ciclul din liniile 4–5 afişează cele două soluţii: 95 şi 45.

Procedura REZOLVĂ-ECUAŢIE-LINIARĂ-MODULARĂ funcţionează după cum urmează. Linia 1 calculează $d = \text{cmmdc}(a, n)$ împreună cu două valori x' şi y', astfel încât $d = ax' + ny'$, demonstrând că x' este o soluţie a ecuaţiei $ax' \equiv d \pmod{n}$. Dacă d nu divide pe b, atunci ecuaţia $ax \equiv b \pmod{n}$ nu are soluţie, conform corolarului 33.21. Linia 2 verifică dacă $d \mid b$; dacă nu, linia 7 ne spune că nu există soluţie. Altfel, linia 3 calculează o soluţie x_0 a ecuaţiei (33.22) conform cu teorema 33.23. Dându-se o soluţie, teorema 33.24 afirmă că celelalte $d - 1$ soluţii pot fi obţinute adăugând multipli lui (n/d) modulo n. Ciclul **pentru** din liniile 4–5 imprimă toate cele d soluţii, începând cu x_0 şi separate la distanţa (n/d), modulo n.

Timpul de execuţie al algoritmului REZOLVĂ-ECUAŢIE-LINIARĂ-MODULARĂ este $O(\lg n + \text{cmmdc}(a, n))$ operaţii aritmetice, deoarece EUCLID-EXTINS necesită $O(\lg n)$ operaţii aritmetice şi fiecare iteraţie a ciclului **pentru** din liniile 4–5 necesită un număr constant de operaţii aritmetice.

Următoarele corolare ale teoremei 33.24 definesc particularizări de interes general.

Corolarul 33.25 Pentru orice $n > 1$, dacă $\text{cmmdc}(a, n) = 1$, atunci ecuaţia $ax \equiv b \pmod{n}$ are o singură soluţie modulo n. ∎

Dacă $b = 1$, x-ul pe care-l căutăm este un ***invers multiplicativ*** al lui a, modulo n.

Corolarul 33.26 Pentru orice $n > 1$, dacă $\text{cmmdc}(a, n) = 1$, atunci ecuaţia

$$ax \equiv 1 \pmod{n} \tag{33.24}$$

are o soluţie unică modulo n. Altfel, nu are soluţie. ∎

Corolarul 33.26 ne permite să folosim notaţia $(a^{-1} \bmod n)$ pentru a ne referi la inversul multiplicativ al lui a modulo n, când a şi n sunt relativi primi. Dacă $\text{cmmdc}(a, n) = 1$, atunci singura soluţie a ecuaţiei $ax \equiv 1 \pmod{n}$ este întregul x returnat de EUCLID-EXTINS, deoarece ecuaţia

$$\text{cmmdc}(a, n) = 1 = ax + ny$$

implică $ax \equiv 1 \pmod{n}$. Astfel, $(a^{-1} \bmod n)$ poate fi calculat eficient folosind EUCLID-EXTINS.

Exerciţii

33.4-1 Găsiţi toate soluţiile ecuaţiei $35x \equiv 10 \pmod{50}$.

33.4-2 Demonstraţi că ecuaţia $ax \equiv ay \pmod{n}$ implică $x \equiv y \pmod{n}$ când $\text{cmmdc}(a, n) = 1$. Demonstraţi necesitatea condiţiei $\text{cmmdc}(a, n) = 1$ indicând un contraexemplu pentru care $\text{cmmdc}(a, n) > 1$.

33.4-3 Se consideră următoarea schimbare în linia 3 a algoritmului REZOLVĂ-ECUAŢIE-LINIARĂ-MODULARĂ:

3 $x_0 \leftarrow x'(b / d) \bmod (n / d)$

Va funcţiona sau nu? Explicaţi de ce.

33.4-4 ⋆ Fie $f(x) \equiv f_0 + f_1 x + \cdots + f_t x^t \pmod{p}$ un polinom de gradul t, având coeficienţi f_i din $\mathbb{Z}_p$, unde p este prim. Spunem că $a \in \mathbb{Z}_p$ este un ***zero*** al lui f dacă $f(a) \equiv 0 \pmod{p}$. Arătaţi că, dacă a este un zero al lui f, atunci $f(x) \equiv (x - a)g(x) \pmod{p}$ pentru un anumit polinom $g(x)$ de grad $t - 1$. Demonstraţi prin inducţie asupra lui t că un polinom $f(x)$ de grad t poate avea cel mult t zerouri distincte modulo un număr prim p.

33.5. Teorema chineză a restului

În jurul anului 100 d.Hr., matematicianul chinez Sun-Tsŭ a rezolvat problema găsirii acelor întregi x care dau resturile 2, 3 şi 2 când se împart la 3, 5 şi, respectiv, 7. O astfel de soluţie este $x = 23$; toate soluţiile sunt de forma $23 + 105k$ pentru numere întregi k arbitrare. "Teorema chineză a restului" furnizează o corespondenţă între un sistem de ecuaţii modulo o mulţime de numere relativ prime două câte două (de exemplu, 3, 5 şi 7) şi o ecuaţie modulo produsul lor (de exemplu, 105).

Teorema chineză a restului are două utilizări majore. Fie întregul $n = n_1 n_2 \cdots n_k$, unde factorii n_i sunt numere relativ prime două câte două. Mai întâi, teorema chineză a restului este o "teoremă de structură" care descrie structura lui $\mathbb{Z}_n$ ca fiind identică cu cea a produsului cartezian $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ cu adunarea şi înmulţirea modulo n_i în a i-a componentă. În al doilea rând, această descriere poate fi, deseori, utilizată pentru a produce algoritmi eficienţi deoarece operarea în fiecare din sistemele $\mathbb{Z}_{n_i}$ poate fi mai eficientă (în termenii operaţiilor pe biţi) decât operarea modulo n.

Teorema 33.27 (Teorema chineză a restului) Fie $n = n_1 n_2 \cdots n_k$, unde n_i sunt relativ prime două câte două. Se consideră corespondenţa

$$a \leftrightarrow (a_1, a_2, \ldots, a_k), \tag{33.25}$$

unde $a \in \mathbb{Z}_n, a_i \in \mathbb{Z}_{n_i}$, iar

$$a_i = a \bmod n_i$$

pentru $i = 1, 2, \ldots, k$. Apoi, transformarea (33.25) este o bijecţie între $\mathbb{Z}_n$ şi produsul cartezian $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$. Operaţiile efectuate asupra elementelor lui $\mathbb{Z}_n$ pot fi efectuate echivalent asupra k-tuplelor corespunzătoare. Astfel, dacă

$$a \leftrightarrow (a_1, a_2, \ldots, a_k), b \leftrightarrow (b_1, b_2, \ldots, b_k),$$

atunci

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k), \tag{33.26}$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k), \tag{33.27}$$

$$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \ldots, a_k b_k \bmod n_k). \tag{33.28}$$

Demonstraţie. Transformarea dintre cele două reprezentări este cât se poate de directă. Trecând de la a la $(a_1, a_2, \ldots, a_k)$, se cer numai k împărţiri. Calculul lui a din intrările $(a_1, a_2, \ldots, a_k)$ este la fel de uşor, utilizând următoarea formulă. Fie $m_i = n/n_i$ pentru $i = 1, 2, \ldots, k$. Să observăm că $m_i = n_1 \cdot n_2 \cdots n_{i-1} \cdot n_{i+1} \cdots n_k$, astfel încât $m_i \equiv 0 \pmod{n_j}$ pentru orice $j \neq i$. Apoi, punând

$$c_i = m_i(m_i^{-1} \mod n_i) \tag{33.29}$$

pentru $i = 1, 2, \ldots, k$, avem

$$a \equiv (a_1 c_1 + a_2 c_2 \cdots + a_k c_k) \pmod{n}. \tag{33.30}$$

Ecuaţia (33.29) este bine definită, deoarece m_i şi n_i sunt relativ prime (prin teorema 33.6) şi corolarul 33.26 implică faptul că $(m_i^{-1} \bmod n_i)$ este definit. Pentru a verifica ecuaţia (33.30), să observăm că $c_j \equiv m_j \equiv 0 \pmod{n_i}$ dacă $j \neq i$, iar $c_i \equiv 1 \pmod{n_i}$. Astfel, avem corespondenţa

$$c_i \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0),$$

vector care are 0-uri peste tot exceptând a i-a componentă, unde are 1. Astfel, c_i formează o "bază" de reprezentare într-un anumit sens. De aceea, pentru fiecare i avem

$$\begin{aligned} a &\equiv a_i c_i &&\pmod{n_i} \\ &\equiv a_i m_i (m_i^{-1} \bmod n_i) &&\pmod{n_i} \\ &\equiv a_i &&\pmod{n_i}. \end{aligned}$$

Întrucât putem transforma în ambele direcţii, corespondenţa este biunivocă. Ecuaţiile (33.26)–(33.28) se deduc direct din exerciţiul 33.1-6, deoarece $x \bmod n_i = (x \bmod n) \bmod n_i$ pentru orice x şi $i = 1, 2, \ldots, k$. ∎

Următoarele corolare vor fi utilizate mai târziu în acest capitol.

Corolarul 33.28 Dacă $n_1, n_2, \ldots, n_k$ sunt relativ prime două câte două şi $n = n_1 n_2 \cdots n_k$, atunci pentru orice întregi $a_1, a_2, \ldots, a_k$, sistemul de ecuaţii

$$x \equiv a_i \pmod{n_i},$$

pentru $i = 1, 2, \ldots, k$, are o soluţie unică modulo n în necunoscuta x.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Figura 33.3 O ilustrare a teoremei chineze a restului pentru $n_1 = 5$ şi $n_2 = 13$. Pentru acest exemplu, $c_1 = 26$ şi $c_2 = 40$. În linia i, coloana j, se arată valoarea lui a, modulo 65, astfel ca $(a \bmod 5) = i$ şi $(a \bmod 13) = j$. Să observăm că linia 0, coloana 0 conţine un 0. Similar, rândul 4, coloana 12 îl conţine pe 64 (echivalentul lui -1). Deoarece $c_1 = 26$, mergând la linia următoare, a se incrementează cu 26. Similar, $c_2 = 40$ înseamnă că, mergând la următoarea coloană, a se măreşte cu 40. Incrementarea lui a cu 1 corespunde deplasării pe diagonală în jos, spre dreapta şi revenirii de jos în sus, de la dreapta spre stânga.

Corolarul 33.29 Dacă $n_1, n_2, \ldots, n_k$ sunt relativ prime două câte două şi $n = n_1 n_2 \cdots n_k$, atunci pentru orice două numere întregi x şi a,

$$x \equiv a \pmod{n_i}$$

pentru $i = 1, 2, \ldots, k$ dacă şi numai dacă

$$x \equiv a \pmod{n}.$$

Ca un exemplu pentru teorema chinezească a restului, presupunem că se dau două ecuaţii

$$a \equiv 2 \pmod{5},$$

$$a \equiv 3 \pmod{13},$$

astfel încât $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$, iar $n_2 = m_1 = 13$, şi vrem să calculăm $a \bmod 65$, deoarece $n = 65$. Întrucât $13^{-1} \equiv 2 \pmod 5$ şi $5^{-1} \equiv 8 \pmod{13}$ avem

$$c_1 = 13\,(2 \bmod 5) = 26,$$

$$c_2 = 5\,(8 \bmod 13) = 40,$$

iar

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &\equiv 52 + 120 \pmod{65} \\ &\equiv 42 \pmod{65}. \end{aligned}$$

O ilustrare a teoremei chineze a restului, modulo 65, se poate vedea în figura 33.3.

Putem lucra direct modulo n sau în reprezentare transformată folosind calcule convenabile modulo n_i, separate. Calculele sunt complet echivalente.

Exerciţii

33.5-1 Găsiţi toate soluţiile ecuaţiilor $x \equiv 4 \pmod 5$ şi $x \equiv 5 \pmod{11}$.

33.5-2 Găsiţi toate numerele întregi x pentru care se obţin resturile 1, 2, 3, 4, 5 când sunt împărţite respectiv la 2, 3, 4, 5, 6.

33.5-3 Argumentaţi că, în conformitate cu definiţiile din teorema 33.27, dacă $\text{cmmdc}(a, n) = 1$, atunci

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_k)).$$

33.5-4 În conformitate cu definiţiile teoremei 33.27, demonstraţi că numărul rădăcinilor ecuaţiei $f(x) \equiv 0 \pmod n$ este egal cu produsul numărului rădăcinilor fiecărei ecuaţii $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}, \ldots, f(x) \equiv 0 \pmod{n_k}$.

33.6. Puterile unui element

Aşa cum este normal să se considere multiplii unui element dat a, modulo n, este la fel de natural să considerăm şirul puterilor lui a, modulo n, unde $a \in \mathbb{Z}_n^\star$:

$$a^0, a^1, a^2, a^3, \ldots, \tag{33.31}$$

modulo n. Valoarea de indice 0 din acest şir este $a^0 \bmod n = 1$, iar a i-a valoare este $a^i \bmod n$. De exemplu, puterile lui 3 modulo 7 sunt

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	$\cdots$

iar puterile lui 2 modulo 7 sunt

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	$\cdots$

În această secţiune vom nota cu $\langle a \rangle$ subgrupul lui $\mathbb{Z}_n^\star$ generat de a, cu $\text{ord}_n(a)$ ("ordinul lui a modulo n"), ordinul lui a în $\mathbb{Z}_n^\star$. De exemplu, $\langle 2 \rangle = \{1, 2, 4\}$ în $\mathbb{Z}_7^\star$, iar $\text{ord}_7(2) = 3$. Utilizând definiţia funcţiei lui Euler $\phi(n)$ ca dimensiune a lui $\mathbb{Z}_n^\star$ (vezi secţiunea 33.3), vom transcrie corolarul 33.19 în notaţia lui $\mathbb{Z}_n^\star$ pentru a obţine teorema lui Euler şi o vom particulariza pentru $\mathbb{Z}_p^\star$, unde p este prim, pentru a obţine teorema lui Fermat.

Teorema 33.30 (Teorema lui Euler) Pentru orice $n > 1$,

$$a^{\phi(n)} \equiv 1 \pmod n \text{ pentru orice } a \in \mathbb{Z}_n^\star. \tag{33.32}$$

Teorema 33.31 (Teorema lui Fermat) Dacă p este prim, atunci

$$a^{p-1} \equiv 1 \pmod n \text{ pentru orice } a \in \mathbb{Z}_n^\star \tag{33.33}$$

Demonstraţie. Din ecuaţia (33.21) rezultă că $\phi(p) = p - 1$ dacă p este prim. ∎

Acest corolar se aplică oricărui element din $\mathbb{Z}_p$, cu excepţia lui 0, deoarece $0 \notin \mathbb{Z}_p^\star$. Totuşi, pentru orice $a \in \mathbb{Z}_p$, avem $a^p \equiv a \pmod{p}$ dacă p este prim.

Dacă $\mathrm{ord}_n(g) = |\mathbb{Z}_n^\star|$, atunci orice element din $\mathbb{Z}_n^\star$ este o putere a lui g modulo n şi spunem că g este o ***rădăcină primitivă*** sau ***generator*** al lui $\mathbb{Z}_n^\star$. De exemplu, 3 este o rădăcină primitivă modulo 7. Dacă $\mathbb{Z}_n^\star$ are o rădăcină primitivă, spunem că grupul $\mathbb{Z}_n^\star$ este ***ciclic***. Omitem demonstraţia următoarei teoreme, care a fost dată de Niven şi Zuckerman [151].

Teorema 33.32 Valorile lui $n > 1$, pentru care $\mathbb{Z}_n^\star$ este ciclic, sunt 2, 4, p^e şi $2p^e$ pentru toate numerele prime impare şi pentru orice întregi e.

Dacă g este o rădăcină primitivă a lui $\mathbb{Z}_n^\star$ şi a este un element arbitrar al lui $\mathbb{Z}_n^\star$, atunci există un z astfel încât $g^z \equiv a \pmod{n}$. Acest z este numit ***logaritmul discret*** sau ***indexul*** lui a modulo n, pentru baza g; notăm această valoare cu $\mathrm{ind}_{n,g}(a)$.

Teorema 33.33 (Teorema logaritmului discret) Dacă g este o rădăcină primitivă a lui $\mathbb{Z}_n^\star$, atunci relaţia $g^x \equiv g^y \pmod{n}$ are loc dacă şi numai dacă are loc $x \equiv y \pmod{\phi(n)}$.

Demonstraţie. Presupunem pentru început că $x \equiv y \pmod{\phi(n)}$. Atunci, $x = y + k\phi(n)$ pentru un anumit întreg k. De aceea,

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} && \pmod{n} \\ &\equiv g^y \cdot (g^{\phi(n)})^k && \pmod{n} \\ &\equiv g^y \cdot 1^k && \pmod{n} \\ &\equiv g^y && \pmod{n} \end{aligned}$$

Invers, presupunem că $g^x \equiv g^y \pmod{n}$. Deoarece şirul puterilor lui g generează orice element al lui $\langle g \rangle$ şi $|\langle g \rangle| = \phi(n)$, corolarul 33.18 implică faptul că şirul de puteri ale lui g este periodic având perioada $\phi(n)$. De aceea, dacă $g^x \equiv g^y \pmod{n}$, atunci trebuie să avem $x \equiv y \pmod{\phi(n)}$. ∎

În unele cazuri, folosirea logaritmilor discreţi poate simplifica deducţiile pentru o relaţie modulară, după cum se arată şi în demonstraţia teoremei următoare.

Teorema 33.34 Dacă p este un număr prim impar şi $e \geq 1$, atunci ecuaţia

$$x^2 \equiv 1 \pmod{p^e} \tag{33.34}$$

are numai două soluţii şi anume $x = 1$ şi $x = -1$.

Demonstraţie. Fie $n = p^e$. Teorema 33.32 implică faptul că $\mathbb{Z}_n^\star$ are o rădăcină primitivă g. Ecuaţia (33.34) poate fi scrisă

$$(g^{\mathrm{ind}_{n,g}(x)})^2 \equiv g^{\mathrm{ind}_{n,g}(1)} \pmod{n}. \tag{33.35}$$

După ce remarcăm că $\mathrm{ind}_{n,g}(1) = 0$, observăm că teorema 33.33 implică faptul că ecuaţia (33.35) este echivalentă cu

$$2 \cdot \mathrm{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)}. \tag{33.36}$$

Pentru a rezolva această ecuaţie în necunoscuta $\text{ind}_{n,g}(x)$, aplicăm metodele din secţiunea 33.4. Fie $d = \text{cmmdc}(2, \phi(n)) = \text{cmmdc}(2, (p-1)p^{e-1}) = 2$ şi observăm că $d \mid 0$, iar din teorema 33.24 deducem că ecuaţia (33.36) are exact $d = 2$, soluţii. De aceea şi ecuaţia (33.34) are exact 2 soluţii care se verifică dacă sunt $x = 1$ şi $x = -1$. ∎

Un număr x este o ***rădăcină netrivială a lui 1 modulo n*** dacă verifică ecuaţia $x^2 \equiv 1 \pmod{n}$ şi x nu este echivalent cu nici una dintre cele două rădăcini "triviale": 1 sau -1 modulo n. De exemplu, 6 este o rădăcină pătrată netrivială a lui 1 modulo 35. Următorul corolar al teoremei 33.34 va fi utilizat la demonstrarea corectitudinii procedurii lui Miller-Rabin de testare a numerelor prime din secţiunea 33.8.

Corolarul 33.35 Dacă există o rădăcină pătrată netrivială a lui 1 modulo n, atunci n este compus.

Demonstraţie. Teorema 33.34 implică faptul că, dacă există o rădăcină pătrată netrivială a lui 1 modulo n, atunci n nu poate fi prim sau o putere a unui număr prim. Deci, n trebuie să fie compus. ∎

Ridicarea la putere prin ridicări repetate la pătrat

O operaţie care apare frecvent în calculele din teoria numerelor este ridicarea unui număr la o putere modulo un alt număr, cunoscută sub numele de ***exponenţiere modulară***. Mai exact, dorim un mod eficient de a calcula $a^b \bmod n$, unde a şi b sunt întregi nenegativi şi n este un întreg pozitiv. Exponenţierea modulară este o operaţie esenţială în multe rutine de testare a proprietăţii de număr prim şi în criptosistemul RSA cu cheie publică. Metoda ***ridicării repetate la pătrat*** rezolvă eficient această problemă utilizând reprezentarea binară a lui b.

Fie $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ reprezentarea binară a lui b. (Mai exact, reprezentarea binară are lungimea de $k+1$ biţi, b_k este cel mai semnificativ bit, iar b_0 este cel mai puţin semnificativ bit.) Următoarea procedură calculează $a^c \bmod n$ mărind pe c prin dublări şi incrementări de la 0 la b.

EXPONENŢIERE-MODULARĂ(a, b, n)

```
1:  c ← 0
2:  d ← 1
3:  fie ⟨b_k, b_{k−1}, ..., b_0⟩ reprezentarea binară a lui b
4:  pentru i ← k, 0, -1 execută
5:     c ← 2c
6:     d ← (d · d) mod n
7:     dacă b_i = 1 atunci
8:        c ← c + 1
9:        d ← (d · a) mod n
10:    returnează d
```

Fiecare exponent calculat succesiv este sau de două ori exponentul precedent, sau cu unu mai mare decât el; reprezentarea binară a lui b este citită de la dreapta spre stânga pentru a decide care operaţie să se realizeze.

Fiecare iteraţie a ciclului utilizează una din identităţile

$$a^{2c} \bmod n = (a^c)^2 \bmod n,$$

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Figura 33.4 Rezultatele lui EXPONENŢIERE-MODULARĂ când se calculează $a^b \pmod n$, pentru $a = 7$, $b = 560 = \langle 1000110000 \rangle$ şi $n = 561$. Sunt arătate valorile după fiecare execuţie a ciclului **pentru**. Rezultatul final este 1.

$$a^{2c+1} \bmod n \quad = \quad a \cdot (a^c)^2 \bmod n,$$

după cum $b_i = 0$, respectiv 1. Folosirea ridicării la pătrat în fiecare iteraţie explică denumirea de "ridicare repetată la pătrat". Imediat după ce bitul b_i este citit şi procesat, valoarea lui c este egală cu prefixul $\langle b_k, b_{k-1}, \ldots, b_i \rangle$ al reprezentării binare a lui b. De exemplu, pentru $a = 7$, $b = 560$ şi $n = 561$, algoritmul calculează şirul de valori modulo 561 ilustrat în figura 33.4; şirul de exponenţi folosiţi se poate observa în rândul c al tabelei.

În realitate, variabila c nu este necesară în algoritm, dar este inclusă pentru scopuri explicative: algoritmul conservă invariantul $d = a^c \bmod n$, în timp ce-l măreşte pe c prin dublări şi incrementări până când $c = b$. Dacă intrările a, b şi n sunt numere reprezentate pe β biţi, atunci numărul total al operaţiilor aritmetice necesare este $O(\beta)$ şi numărul total al operaţiilor pe biţi necesare este $O(\beta^2)$.

Exerciţii

33.6-1 Întocmiţi un tabel care să arate ordinul fiecărui element din $\mathbb{Z}_{11}^\star$. Găsiţi cea mai mică rădăcină primitivă g şi alcătuiţi o tabelă cu $\text{ind}_{11,g}(x)$ pentru orice $x \in \mathbb{Z}_{11}^\star$.

33.6-2 Daţi un algoritm de exponenţiere modulară care să examineze toţi biţii din reprezentarea lui b de la dreapta spre stânga, şi nu de la stânga spre dreapta.

33.6-3 Explicaţi cum se poate calcula $a^{-1} \bmod n$ pentru orice $a \in \mathbb{Z}_n^\star$ utilizând procedura EXPONENŢIERE-MODULARĂ, presupunând că se cunoaşte $\phi(n)$.

33.7. Criptosistemul RSA cu cheie publică

Un criptosistem cu cheie publică poate fi utilizat pentru a cripta mesaje trimise între două părţi care comunică între ele, astfel încât o persoană străină care interceptează mesajele criptate să nu le poată decodifica. De asemenea, un criptosistem cu cheie publică permite unei părţi să adauge o "semnătură digitală" nefalsificabilă la sfârşitul unui mesaj electronic. O astfel de semnătură este versiunea electronică a unei semnături scrise cu mâna pe un document oficial scris pe hârtie. Ea poate fi verificată uşor de către oricine, nu poate fi falsificată de nimeni, totuşi îşi pierde validitatea dacă cel puţin un bit din mesaj este alterat. De aceea, ea furnizează atât autenticitatea identităţii semnatarului cât şi a conţinutului mesajului semnat. Este un instrument perfect pentru contractele de afaceri semnate electronic, pentru cecurile electronice, comenzile electronice de cumpărare şi alte comunicaţii electronice care trebuie autentificate.

Criptosistemul RSA cu cheie publică se bazează pe diferenţa impresionantă dintre uşurinţa de a găsi numere prime mari şi dificultatea de a factoriza produsul a două numere prime mari. În secţiunea 33.8 se descrie o procedură eficientă pentru a găsi numere prime mari, iar în secţiunea 33.9 se discută problema factorizării întregilor mari.

Criptosisteme cu cheie publică

Într-un criptosistem cu cheie publică, fiecare participant are atât o ***cheie publică*** cât şi o ***cheie secretă***. Fiecare cheie este un fragment de informaţie. De exemplu, în criptosistemul RSA, fiecare cheie constă dintr-o pereche de numere întregi. Numele "Alice" şi "Bob" se utilizează, tradiţional, în exemplele de criptografie; notăm cheile lor publice şi secrete cu P_A, S_A pentru Alice şi P_B, S_B pentru Bob.

Fiecare participant îşi creează propria cheie publică şi secretă. Fiecare îşi păstrează în secret cheia sa secretă, dar poate să-şi deconspire cheia publică oricui sau chiar să o facă publică. De fapt, putem presupune că o cheie publică aparţinând unei persoane este disponibilă într-un director public, astfel încât orice participant poate obţine uşor cheia publică a oricărui alt participant.

Cheile publice şi secrete specifică funcţii care pot fi aplicate oricărui mesaj. Notăm prin $\mathcal{D}$ mulţimea mesajelor permise. De exemplu, $\mathcal{D}$ ar putea fi mulţimea tuturor şirurilor de biţi de lungime finită. Se cere ca ambele chei să specifice funcţii unu-la-unu (bijective) de la $\mathcal{D}$ la $\mathcal{D}$. Funcţia care corespunde cheii publice P_A a lui Alice se notează cu $P_A()$, iar cea care corespunde cheii secrete S_A cu $S_A()$. Funcţiile $P_A()$ şi $S_A()$ sunt permutări ale lui $\mathcal{D}$. Presupunem că funcţiile $P_A()$ şi $S_A()$ se pot calcula eficient cunoscând cheile corespunzătoare P_A şi S_A.

Cheia publică şi cea secretă pentru fiecare participant, sunt o "pereche asortată", în sensul că ele specifică funcţii inverse una celeilalte. Adică,

$$M = S_A(P_A(M)), \tag{33.37}$$

$$M = P_A(S_A(M)) \tag{33.38}$$

pentru orice mesaj $M \in \mathcal{D}$. Transformându-l succesiv pe M cu ajutorul celor două chei P_A şi S_A, în orice ordine, se revine la mesajul M.

Într-un criptosistem cu cheie publică, este esenţial ca nimeni, în afară de Alice, să nu poată calcula funcţia $S_A()$ într-un timp rezonabil. Secretul mesajului criptat şi expediat lui Alice şi autenticitatea semnăturii digitale a lui Alice se bazează pe presupunerea că numai Alice este în stare să calculeze pe $S_A()$. Această cerinţă impune ca Alice să păstreze S_A secret; dacă nu face acest lucru, ea îşi pierde unicitatea şi criptosistemul nu o poate înzestra cu facilităţi unice. Presupunerea că numai Alice poate calcula $S_A()$ trebuie să se menţină chiar dacă cineva cunoaşte P_A şi poate calcula eficient $P_A()$, inversa funcţiei $S_A()$. Dificultatea majoră în proiectarea unui criptosistem funcţionabil cu cheie publică constă în figurarea modului de a crea un sistem în care putem deconspira o transformare $P_A()$, fără a deconspira cum se calculează transformarea inversă $S_A()$ corespunzătoare.

Într-un criptosistem cu cheie publică, criptarea funcţionează după cum urmează: presupunem că Bob doreşte să-i trimită lui Alice un mesaj M criptat, astfel încât el va arăta ca o vorbire neinteligibilă pentru o persoană care trage cu urechea. Scenariul pentru trimiterea mesajului se desfăşoară după cum urmează:

- Bob obţine cheia publică P_A a lui Alice (dintr-un director public sau direct de la Alice).

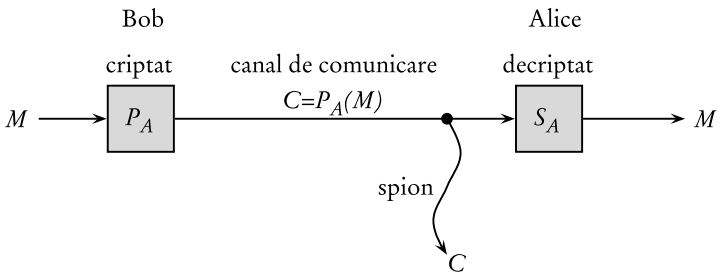


Figura 33.5 Criptarea într-un sistem cu cheie publică. Bob criptează mesajul M utilizând cheia publică P_A a lui Alice şi transmite textul cifrat care a rezultat $C = P_A(M)$ lui Alice. O persoană străină care capturează transmisia textului cifrat nu obţine nici o informaţie despre M. Alice îl recepţionează şi-l decriptează folosind cheia ei secretă pentru a obţine mesajul original $M = S_A(C)$.

- Bob calculează ***textul cifrat*** $C = P_A(M)$ care corespunde mesajului M şi trimite C lui Alice.
- Când Alice recepţionează textul cifrat C, aplică cheia ei secretă S_A pentru a obţine mesajul original $M = S_A(C)$.

Figura 33.5 ilustrează acest proces. Deoarece $S_A()$ şi $P_A()$ sunt funcţii inverse, Alice îl poate calcula pe M din C. Deoarece numai Alice este în stare să-l calculeze pe $S_A()$, numai ea îl poate calcula pe M din C. Criptarea lui M folosind $P_A()$ protejează dezvăluirea lui M faţă de oricine, exceptând-o pe Alice.

Semnăturile digitale sunt la fel de uşor de implementat într-un criptosistem cu cheie publică. Presupunem acum că Alice doreşte să-i trimită lui Bob un răspuns M', semnat digital. Scenariul pentru semnătură digitală continuă după cum urmează:

- Alice calculează ***semnătura digitală*** σ a ei pentru mesajul M' folosind cheia ei secretă S_A şi ecuaţia $\sigma = S_A(M')$.
- Alice trimite lui Bob perechea (M', σ) mesaj/semnătură.
- Când Bob recepţionează (M', σ), poate verifica faptul că acesta este de la Alice, folosind cheia publică a lui Alice pentru a verifica relaţia $M' = P_A(\sigma)$. (Probabil că M' conţine numele lui Alice, astfel încât Bob ştie a cui cheie publică să o folosească). Dacă relaţia este satisfăcută, Bob trage concluzia că mesajul M' a fost într-adevăr semnat de Alice. Dacă relaţia nu este satisfăcută, Bob trage concluzia că M' sau semnătura digitală σ au fost deteriorate prin erori de transmisie sau că perechea (M', σ) este un fals.

Figura 33.6 ilustrează acest proces. Deoarece o semnătură digitală furnizează autenticitatea atât a identităţii semnatarului, cât şi a conţinutului mesajului semnat, ea este analogă cu o semnătură scrisă cu mâna la sfârşitul unui document scris.

O proprietate importantă a unei semnături digitale este faptul că este verificabilă de oricine are acces la cheia publică a semnatarului. Un mesaj semnat poate fi verificat de mai multe ori (de mai multe părţi). De exemplu, mesajul ar putea fi un cec electronic de la Alice la Bob. După ce Bob verifică semnătura lui Alice de pe cec, el poate preda cecul la banca sa, care apoi poate verifica din nou semnătura şi poate efectua transferul corespunzător de capital.

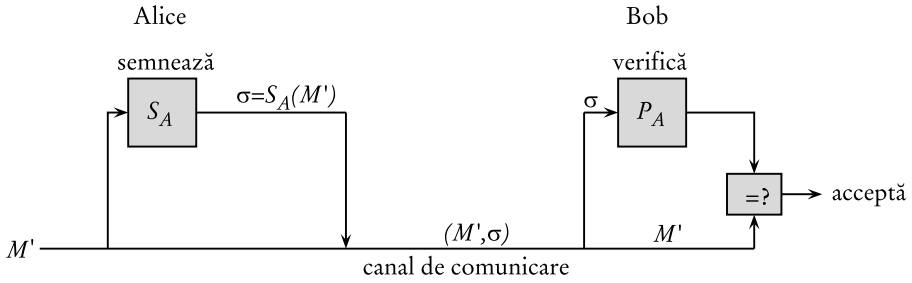


Figura 33.6 Semnături digitale într-un sistem cu cheie publică. Alice semnează mesajul M' folosind semnătura ei digitală $\sigma = S_A(M')$. Îi transmite lui Bob perechea (M', σ) mesaj/semnătură, el o verifică testând relaţia $M' = P_A(\sigma)$. Dacă relaţia este satisfăcută, acceptă pe (M', σ) drept mesaj care a fost semnat de Alice.

Observăm că un mesaj semnat nu este criptat; mesajul este "în clar" şi nu este protejat contra falsificării. Compunând protocoalele de mai sus pentru criptare şi semnături, putem crea mesaje care să fie atât semnate cât şi criptate. Semnatarul adaugă întâi semnătura sa digitală la mesaj şi apoi criptează, rezultând perechea mesaj/semnătură cu cheie publică a destinatarului avut în vedere. Receptorul decriptează cu cheia sa secretă mesajul primit pentru a obţine atât mesajul original cât şi semnătura lui digitală. El poate, apoi, verifica semnătura utilizând cheia publică a semnatarului. Procesul combinat este corespunzător practicii folosite la documentele scrise, aceea de a semna documentul, apoi, a sigila plicul în care acesta a fost introdus şi care poate fi deschis numai de către destinatar.

Criptosistemul RSA

În ***criptosistemul RSA cu cheie publică***, un participant creează cheia sa publică şi secretă prin următoarea procedură:

1. Se selectează aleator două numere prime mari p şi q. Acestea ar putea avea, de exemplu, 100 de cifre zecimale.
2. Se calculează n prin relaţia $n = pq$.
3. Se selectează un număr impar mic e care este relativ prim cu $\phi(n)$ şi care, conform relaţiei (33.20), este $(p-1)(q-1)$.
4. Se calculează d ca fiind inversul multiplicativ al lui e modulo $\phi(n)$. (Corolarul 33.26 garantează că d există şi că acesta este unic).
5. Se declară perechea $P = (e, n)$ drept ***cheie RSA publică***.
6. Se menţine secretă perechea $S = (d, n)$ care este ***cheie RSA secretă***.

Pentru această schemă, domeniul $\mathcal{D}$ este mulţimea $\mathbb{Z}_n$. Transformarea mesajului M asociat unei chei publice $P = (e, n)$ este

$$P(M) = M^e \pmod{n}. \tag{33.39}$$

Transformarea textului cifrat C asociat unei chei secrete $S = (d, n)$ este:

$$S(C) = C^d \pmod{n}. \tag{33.40}$$

Aceste relaţii se referă atât la criptare, cât şi la semnături. Pentru a crea o semnătură, semnatarul aplică cheia sa secretă mesajului de semnat şi nu textului cifrat. Pentru a verifica o semnătură, cheia publică a semnatarului i se aplică ei şi nu mesajului criptat.

Operaţiile pentru cheie publică şi cheie secretă pot fi implementate folosind procedura EXPONENŢIERE-MODULARĂ descrisă în secţiunea 33.6. Pentru a analiza timpul de execuţie al acestor operaţii, să presupunem că ambele chei, cea publică (e, n) şi cea secretă (d, n), satisfac relaţiile $\log_2 e = O(1)$, $\log_2 d = \log_2 n = \beta$. Atunci, aplicarea unei chei publice necesită $O(1)$ înmulţiri modulare şi se utilizează $O(\beta^2)$ operaţii pe biţi. Aplicarea unei chei secrete necesită $O(\beta)$ înmulţiri modulare şi $O(\beta^3)$ operaţii pe biţi.

Teorema 33.36 (Corectitudinea lui RSA) Relaţiile RSA (33.39) şi (33.40) definesc transformările inverse ale lui $\mathbb{Z}_n$, care satisfac relaţiile (33.37) şi (33.38).

Demonstraţie. Din relaţiile (33.39) şi (33.40), pentru orice $M \in \mathbb{Z}_n$, avem

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n}.$$

Deoarece e şi d sunt inverse multiplicative modulo $\phi(n) = (p-1)(q-1)$,

$$ed = 1 + k(p-1)(q-1)$$

pentru un anumit întreg k. Dar atunci, dacă $M \not\equiv 0 \pmod{p}$, avem (folosind teorema 33.31)

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M(1)^{k(q-1)} && \pmod{p} \\ &\equiv M && \pmod{p}. \end{aligned}$$

De asemenea, $M^{ed} \equiv M \pmod{p}$ dacă $M \equiv 0 \pmod{p}$. Astfel,

$$M^{ed} \equiv M \pmod{p}$$

pentru orice M. Similar,

$$M^{ed} \equiv M \pmod{q}$$

pentru orice M. Astfel, pe baza corolarului 33.29 al teoremei chinezeşti a restului,

$$M^{ed} \equiv M \pmod{n}$$

pentru orice M. ∎

Securitatea criptosistemului rezidă, în mare parte, în dificultatea de a factoriza numere întregi mari. Dacă adversarul poate factoriza modulul n printr-o cheie publică, atunci poate obţine cheia secretă din cea publică, utilizând cunoştinţele despre factorii p şi q în acelaşi mod în care le-a utilizat creatorul cheii publice. Astfel, dacă factorizarea numerelor întregi mari este uşoară, atunci spargerea criptosistemului RSA este simplă. Afirmaţia inversă, şi anume că, dacă factorizarea numerelor întregi mari este dificilă, atunci spargerea lui RSA este dificilă, este nedemonstrabilă. Totuşi, după un deceniu de cercetări, nu a fost găsită nici o metodă mai simplă de a sparge

criptosistemul RSA cu cheie publică, decât factorizarea modulului n. Aşa cum vom vedea în secţiunea 33.9, factorizarea numerelor întregi mari este surprinzător de dicifilă. Alegând aleator şi înmulţind două numere prime de 100 de cifre, se poate crea o cheie publică ce nu poate fi "spartă" cu tehnologia curentă într-un timp rezonabil. În absenţa unui salt fundamental în proiectarea algoritmilor din teoria numerelor, criptosistemul RSA este capabil să furnizeze un grad înalt de securitate în aplicaţii.

În vederea obţinerii securităţii cu criptosistemele RSA, este totuşi necesar să se lucreze cu numere întregi care să aibă 100–200 de cifre, deoarece factorizarea întregilor mai mici nu este impractică. În particular, trebuie să fim în stare să găsim eficient numere prime mari în vederea creării cheilor de lungime necesară. Această problemă este prezentată în secţiunea 33.8.

Pentru eficienţă, RSA este, adesea, utilizat într-un mod "hibrid" sau "controlat-de-cheie" cu criptosistem rapid având cheie nepublică. Cu un astfel de sistem, cheile de criptare şi decriptare sunt identice. Dacă Alice doreşte să-i trimită lui Bob un mesaj confidenţial M lung, selectează o cheie aleatoare K pentru criptosistemul cu cheie nepublică, criptează mesajul M cu ajutorul cheii K şi obţine textul cifrat C. Textul C este tot atât de lung ca M, dar cheia K este cât se poate de scurtă. Apoi, criptează K folosind cheia RSA publică a lui Bob. Deoarece K este scurtă, calculul lui $P_B(K)$ este rapid (mult mai rapid decât calculul lui $P_B(M)$). Apoi, $(C, P_B(K))$ este transmis lui Bob, care decriptează $P_B(K)$ pentru a-l obţine pe K şi, apoi, îl foloseşte pe K pentru a-l decripta pe C, obţinându-l pe M.

O altă concepţie hibridă similară este des folosită pentru a realiza semnături digitale eficiente. În această concepţie, RSA este înzestrat cu o ***funcţie univocă de dispersie*** publică h, o funcţie care este uşor da calculat, dar pentru care nu este simplu, din punctul de vedere al calculului, să se găsească două mesaje M şi M' astfel încât $h(M) = h(M')$. Valoarea $h(M)$ este o "amprentă" prescurtată (de exemplu, de 128 biţi) a mesajului M. Dacă Alice doreşte să semneze un mesaj M, aplică întâi h lui M pentru a obţine amprenta $h(M)$, pe care apoi o semnează cu cheia ei secretă. Va trimite $(M, S_A(h(M)))$ lui Bob ca versiune a lui M, semnată de ea. Bob poate verifica semnătura prin calculul lui $h(M)$ pe de o parte, iar pe de alta, aplicând P_A la $S_A(h(M))$, obţinând astfel $h(M)$. Deoarece nimeni nu poate crea două mesaje cu aceeaşi amprentă, este imposibil să se altereze un mesaj semnat şi să se păstreze validitatea semnăturii.

În final, să observăm că folosirea ***certificatelor*** uşurează distribuirea cheilor publice. De exemplu, să presupunem că există o "autoritate de încredere" T a cărei cheie publică este cunoscută de oricine. Alice poate obţine de la T un mesaj semnat (certificatul ei) afirmând că "Alice are cheia publică P_A". Acest certificat se "auto-identifică" deoarece oricine cunoaşte pe P_T. Alice poate include certificatul ei cu mesajul semnat de ea, astfel încât destinatarul are imediat disponibilă cheia publică a lui Alice în vederea verificării semnăturii. Deoarece cheia ei a fost semnată de T, destinatarul ştie că, într-adevăr cheia lui Alice este a lui Alice.

Exerciţii

33.7-1 Se consideră o cheie RSA cu $p = 11$, $q = 29$, $n = 319$ şi $e = 3$. Ce valoare ar trebui folosită pentru d în cheia secretă? Care este criptarea mesajului $M = 100$?

33.7-2 Demonstraţi că, dacă exponentul public al lui Alice este 3 şi un adversar obţine exponentul secret d al lui Alice, atunci adversarul poate factoriza modulul n al lui Alice într-un timp polinomial în numărul de biţi din n. (Deşi nu se cere să se demonstreze, putem fi interesaţi

să ştim că acest rezultat rămâne valabil şi în cazul în care condiţia $e = 3$ este eliminată. Vezi Miller [147].)

33.7-3 ⋆ Arătaţi că RSA este multiplicativ, în sensul că

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2)(\bmod n).$$

Să se utilizeze acestă proprietate pentru a demonstra că, dacă un adversar are o procedură care ar putea decripta eficient 1 procent din mesajele alese aleator din $\mathbb{Z}_n$ şi care sunt criptate cu P_A, atunci ar putea utiliza un algoritm probabilistic pentru a decripta, cu probabilitate mare, orice mesaj criptat cu P_A.

33.8. Testul de primalitate

În această secţiune considerăm problema găsirii numerelor prime mari. Începem cu o discuţie despre densitatea numerelor prime, continuând cu examinarea unei concepţii plauzibile (dar incomplete) pentru testarea proprietăţii de număr prim şi apoi prezentăm un test aleator efectiv pentru numere prime, datorat lui Miller şi Rabin.

Densitatea numerelor prime

Pentru multe aplicaţii (cum este criptarea), avem nevoie să găsim numere prime mari "aleatoare". Din fericire, numerele prime mari nu sunt prea rare, aşa că nu este prea costisitor în timp să testăm aleator întregi de dimensiune corespunzătoare, până când găsim un număr prim. ***Funcţia*** $\pi(n)$ ***de distribuţie a numerelor prime*** specifică numărul numerelor prime care sunt mai mici sau egale cu n. De exemplu, $\pi(10) = 4$, deoarece există 4 numere prime mai mici sau egale cu 10 şi anume 2, 3, 5 şi 7. Teorema numărului numerelor prime oferă o aproximaţie utilă pentru $\pi(n)$.

Teorema 33.37 (Teorema numărului numerelor prime)

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1.$$

■

Aproximaţia $n/\ln n$ dă o estimare rezonabil de precisă pentru $\pi(n)$, chiar şi pentru n mic. De exemplu, ea diferă cu mai puţin de 6% pentru $n = 10^9$, unde $\pi(n) = 50847534$ şi $n/\ln n$ $=48254942$. (În teoria numerelor, 10^9 este considerat un număr mic).

Putem folosi teorema numărului numerelor prime pentru a estima că probabilitatea ca un număr întreg n ales la întâmplare să fie număr prim este $1/\ln n$. Astfel, va trebui să examinăm $\ln n$ numere întregi apropiate de n, alese aleator, pentru a găsi un număr prim care să fie de aceeaşi lungime ca n. De exemplu, pentru a găsi un număr prim de 100 de cifre, s-ar cere să se testeze dacă sunt prime aproximativ $\ln 10^{100} \approx 230$ numere de 100 de cifre alese aleator. (Acest număr poate fi înjumătăţit alegând numai întregi impari).

În cele ce urmează, în această secţiune, vom considera problema determinării dacă n (număr întreg impar, mare) este prim sau nu. Din punctul de vedere al notaţiilor, este convenabil să presupunem că n are descompunerea în numere prime

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \tag{33.41}$$

unde $r \geq 1$ şi $p_1, p_2, \ldots, p_r$ sunt factorii primi ai lui n. Desigur, n este prim dacă şi numai dacă $r = 1$ şi $e_1 = 1$.

O concepţie simplă pentru problema testării numerelor prime este ***împărţirea de probă***. Se încearcă să se dividă n prin fiecare întreg 2, 3, $\ldots, \lfloor \sqrt{n} \rfloor$. (Din nou, întregii pari mai mari decât 2 pot fi omişi.) Este uşor să se observe că n este prim dacă şi numai dacă nici unul din divizorii de probă nu-l divide pe n. Presupunând că fiecare împărţire de probă necesită un timp constant, timpul de execuţie în cazul cel mai defavorabil este $\Theta(\sqrt{n})$, care este exponenţial în raport cu lungimea lui n. (Amintim că, dacă n are o reprezentare binară pe β biţi, atunci $\beta = \lceil \lg(n+1) \rceil$ şi astfel $\sqrt{n} = \Theta(2^{\beta/2})$.) Astfel, împărţirea de probă funcţionează bine numai dacă n este foarte mic sau are un factor prim mic. Când împărţirea de probă funcţionează, are avantajul că determină nu numai faptul că n este prim sau compus, dar dacă n este compus, determină unul din factorii primi ai lui.

În această secţiune suntem interesaţi numai în a determina dacă un număr dat n este prim sau compus şi nu suntem interesaţi de găsirea factorizării lui în numere prime. Aşa cum vom vedea în secţiunea 33.9, determinarea factorizării în numere prime a unui număr dat este laborioasă din punctul de vedere al calculelor. Este surprinzător, dar este mult mai uşor să afirmi dacă un număr este prim sau nu, decât să determini factorizarea în numere prime dacă el nu este prim.

Testul de pseudoprimalitate

Considerăm acum o metodă de testare a proprietăţii de număr prim care "aproape funcţionează" şi care, de fapt, este destul de bună pentru multe aplicaţii practice. O versiune rafinată a metodei, care elimină micul ei neajuns, va fi prezentată mai târziu. Prin $\mathbb{Z}_n^+$ notăm elementele nenule ale lui $\mathbb{Z}_n$:

$$\mathbb{Z}_n^+ = \{1, 2, \ldots, n-1\}.$$

Dacă n este prim, atunci $\mathbb{Z}_n^+ = \mathbb{Z}_n^\star$.

Spunem că n este ***pseudoprim cu baza a*** dacă n este compus şi

$$a^{n-1} \equiv 1 \pmod{n}. \tag{33.42}$$

Teorema lui Fermat (teorema 33.31) implică faptul că dacă n este prim, atunci n satisface relaţia (33.42) pentru orice a din $\mathbb{Z}_n^+$. Astfel, dacă putem găsi un $a \in \mathbb{Z}_n^+$ oarecare astfel încât n să *nu* satisfacă relaţia (33.42), atunci cu siguranţă n este compus. În mod surprinzător şi inversa este *aproape* întotdeauna adevărată, astfel încât acest criteriu formează un test aproape perfect al faptului că un număr este prim. Verificăm dacă n satisface relaţia (33.42) pentru a =2. Dacă nu, declarăm că n este compus. Altfel, presupunem că n este prim (în realitate, tot ce ştim este că n este fie prim, fie pseudoprim cu baza 2).

Următoarea procedură realizează, în acest fel, testul proprietăţii de număr prim a lui n. Ea utilizează procedura EXPONENŢIERE-MODULARĂ din secţiunea 33.6. Intrarea n se presupune a fi un întreg mai mare decât 2.

PSEUDOPRIM(n)
1: **dacă** EXPONENŢIERE-MODULARĂ($2, n-1, n$) $\not\equiv 1 \pmod{n}$ **atunci**
2: **returnează** COMPUS ▷ Definitiv.
3: **altfel**
4: **returnează** PRIM ▷ Se speră!

Această procedură poate genera erori, dar numai de un singur tip. Adică, dacă ea afirmă că n este compus, atunci aceasta este totdeauna corect. Dacă ea spune că n este prim, atunci ea poate face o eroare numai dacă n este pseudoprim cu baza 2.

Cât de des poate să apară această eroare? Surprinzător de rar. Există numai 22 de valori ale lui n mai mici decât 10000 pentru care ea este eronată; primele patru astfel de erori sunt 341, 561, 645 şi 1105. Se poate arăta că probabilitatea, ca acest program să genereze o eroare pentru o alegere aleatoare a unui număr care are o reprezentare pe β biţi, tinde la zero când $\beta \to \infty$. Utilizând estimarea mai precisă, dată de Pomerance [157], a numărului de pseudoprime cu baza 2 pentru o dimensiune dată, putem estima că un număr de 50 de cifre ales aleator, care este considerat prim prin procedura de mai sus, are mai puţin decât o şansă la un milion să fie pseudoprim cu baza 2, iar un număr de 100 de cifre ales aleator, care este considerat prim are mai puţin de o şansă la 10^{13} să fie pseudoprim cu baza 2.

Din nefericire, nu putem elimina toate erorile prin simpla testare a relaţiei (33.42) pentru un al doilea număr de bază, să zicem 3, deoarece există numere întregi compuse n care satisfac relaţia (33.42) pentru orice $a \in \mathbb{Z}_n^\star$. Aceşti întregi se numesc ***numere Carmichael***. Primele trei numere Carmichael sunt 561, 1105 şi 1729. Numerele Carmichael sunt extrem de rare; de exemplu, există numai 255 astfel de numere mai mici decât 100000000. Exerciţiul 33.8-2 ajută la explicarea acestui fapt.

În continuare, arătăm cum se îmbunătăţeşte testul de număr prim, astfel încât acesta să nu fie înşelat de numere Carmichael.

Testul aleator de primalitate al lui Miller-Rabin

Testul de număr prim al lui Miller-Rabin depăşeşte problemele testului simplu PSEUDOPRIM prin două modificări:

- Încearcă să aleagă aleator diferite valori ale bazei a în loc de o singură valoare.
- În timp ce calculează fiecare exponenţiere modulară, stabileşte dacă nu cumva a fost descoperită o rădăcină netrivială a lui 1 modulo n. Dacă este aşa, se opreşte cu ieşirea COMPUS. Corolarul 33.35 justifică detectarea numerelor compuse în acest fel.

Algoritmul pentru testul de număr prim al lui Miller-Rabin este prezentat mai jos. Intrarea $n > 2$ este numărul impar care se testează dacă este prim, iar s este valoarea bazei aleasă aleator din $\mathbb{Z}_n^+$ pentru a se face testul. Algoritmul utilizează generatorul de numere aleatoare RANDOM din secţiunea 8.3: RANDOM($1, n-1$) returnează un număr a ales aleator care satisface $1 \le a \le n-1$. Algoritmul utilizează o procedură auxiliară PROBA, astfel încât PROBA(a, n) este ADEVĂRAT dacă şi numai dacă a este o "probă" pentru proprietatea de compus a lui n adică, dacă este posibil să se demonstreze (în maniera în care vom vedea) utilizând a, că n este compus. Testul PROBA(a, n) este similar cu (dar mult mai eficient decât) testul

$$a^{n-1} \not\equiv 1 \pmod{n}$$

care a stat (utilizând $a = 2$) la baza algoritmului PSEUDOPRIM. Mai întâi prezentăm şi justificăm construcţia lui PROBA şi, apoi, arătăm cum se utilizează aceasta în testul Miller-Rabin.

```
PROBA(a, n)
 1: fie ⟨b_k, b_{k-1}, ..., b_0⟩ reprezentarea binară a lui n-1
 2: d ← 1
 3: pentru i ← k, 0, −1 execută
 4:    x ← d
 5:    d ← (d · d) mod n
 6:    dacă d = 1 şi x ≠ 1 şi x ≠ n − 1 atunci
 7:       returnează ADEVĂRAT
 8:    dacă b_i = 1 atunci
 9:       d ← (d · a) mod n
10: dacă d ≠ 1 atunci
11:    returnează ADEVĂRAT
12: returnează FALS
```

Acest algoritm pentru PROBA se bazează pe procedura EXPONENŢIERE-MODULARĂ. Linia 1 determină reprezentarea binară a lui $n - 1$, care va fi folosită la ridicarea lui a la puterea $n - 1$. Liniile 3–9 calculează pe d ca fiind $a^{n-1} \bmod n$. Metoda utilizată este identică cu cea utilizată de EXPONENŢIERE-MODULARĂ. Ori de câte ori se face un pas de ridicare la pătrat în linia a 5-a, liniile 6–7 verifică dacă nu s-a descoperit chiar o rădăcină pătrată netrivială a lui 1. Dacă este aşa, algoritmul se opreşte şi returnează ADEVĂRAT. Liniile 10–11 returnează FALS dacă valoarea calculată pentru $a^{n-1} \bmod n$ nu este egală cu 1, aşa cum procedura PSEUDOPRIM returnează COMPUS, în acelaşi caz.

Acum argumentăm că, dacă PROBA(a, n) returnează ADEVĂRAT, atunci, se poate construi folosind a o demonstraţie a faptului că n este compus.

Dacă PROBA returnează ADEVĂRAT în linia 11, atunci algoritmul a găsit că $d = a^{n-1} \bmod n \neq 1$. Dacă n este prim, totuşi, pe baza teoremei lui Fermat (teorema 33.31), avem $a^{n-1} \equiv 1 \pmod{n}$ pentru orice $a \in \mathbb{Z}_n^+$. De aceea, n nu poate fi prim, iar ecuaţia $a^{n-1} \bmod n \neq 1$ este o demonstraţie a acestui fapt.

Dacă PROBA returnează ADEVĂRAT în linia 7, atunci s-a descoperit că x este o rădăcină pătrată netrivială a lui 1 modulo n, deoarece avem $x \not\equiv \pm 1 \pmod{n}$ şi $x^2 \equiv 1 \pmod{n}$. Corolarul 33.35 afirmă că, numai dacă n este compus, poate să existe o rădăcină pătrată netrivială a lui 1 modulo n, deci o demostraţie a faptului că x este o rădăcină pătrată netrivială a lui 1 modulo n demonstrează faptul că n este compus.

Aceasta completează demonstraţia noastră asupra corectitudinii lui PROBA. Dacă apelul PROBA(a, n) returnează ADEVĂRAT, atunci n este, cu siguranţă compus, iar o demonstraţie a faptului că n este compus poate fi uşor determinată din a şi n. Acum, examinăm testul de număr prim al lui Miller-Rabin bazat pe utilizarea lui PROBA.

```
MILLER-RABIN(n, s)
1: pentru j ← 1, s execută
2:    a ←RANDOM(1, n − 1)
3:    dacă PROBA(a, n) atunci
4:       returnează COMPUS        ▷ Definitiv
5: returnează PRIM                ▷ Aproape sigur
```

Procedura MILLER-RABIN este o căutare probabilistică pentru a demonstra că n este compus. Ciclul principal (care începe în linia 1) găseşte s valori aleatoare pentru a din $\mathbb{Z}_n^+$ (linia 2). Dacă una din valorile alese pentru a este o probă pentru proprietatea de număr compus a lui n, atunci MILLER-RABIN returnează COMPUS în linia 4. O astfel de ieşire este totdeauna corectă, datorită corectitudinii lui PROBA. Dacă din s încercări nu se găseşte nici o probă, algoritmul MILLER-RABIN presupune că acest lucru se datorează faptului că nu s-a găsit nici o probă şi returnează că n este prim. Vom vedea că această ieşire este probabil corectă dacă s este destul de mare, dar există o mică şansă ca procedura să fie nenorocoasă în alegerea lui a şi ca probele să existe chiar dacă nu s-a găsit nici una.

Pentru a ilustra procedura lui MILLER-RABIN, fie n numărul Charmichael 561. Presupunând că $a = 7$ este aleasă ca bază, figura 33.4 arată că PROBA a descoperit o rădăcină pătrată netrivială a lui 1 în ultimul pas de ridicare la pătrat, deoarece $a^{280} \equiv 67 \pmod{n}$ şi $a^{560} \equiv 1 \pmod{n}$. De aceea, $a = 7$ este o probă în stabilirea faptului că n este compus, PROBA$(7, n)$ returnează ADEVĂRAT, iar MILLER-RABIN, COMPUS.

Dacă n este un număr care are reprezentarea binară pe β biţi, MILLER-RABIN necesită $O(s\beta)$ operaţii aritmetice şi $O(s\beta^3)$ operaţii pe biţi, deoarece se cer nu mai mult decât s exponenţieri modulare.

Rata de eroare a testului de primalitate Miller-Rabin

Dacă algoritmul MILLER-RABIN returnează PRIM, atunci există o mică şansă de eroare. Spre deosebire de PSEUDOPRIM, şansa de eroare nu depinde de n; nu există intrări defavorabile pentru această procedură. Mai degrabă depinde de dimensiunea lui s şi de "norocul avut" la alegerea valorilor bazei a. De asemenea, deoarece fiecare test este mai riguros decât un simplu test al ecuaţiei (33.42), ne putem aştepta, pe baza principiilor generale, ca rata erorii să fie mai mică pentru numere întregi n alese aleator. Următoarea teoremă prezintă un argument mai convingător.

Teorema 33.38 Dacă n este un număr compus impar, atunci numărul de probe pentru proprietatea de număr compus a lui n este de cel puţin $(n-1)/2$.

Demonstraţie. Demonstraţia constă în a arăta că numărul de nonprobe nu este mai mare decât $(n-1)/2$, ceea ce implică teorema.

Să observăm că orice nonprobă trebuie să fie un membru al lui $\mathbb{Z}_n^\star$, deoarece orice nonprobă a satisface $a^{n-1} \equiv 1 \pmod{n}$, în plus, dacă $\text{cmmdc}(a, n) = d > 1$ atunci, conform corolarului 33.21, nu există nici o soluţie x a ecuaţiei $ax \equiv 1 \pmod{n}$. (În particular, $x = a^{n-2}$ nu este o soluţie.) Astfel, orice membru al lui $\mathbb{Z}_n - \mathbb{Z}_n^\star$ este o probă a proprietăţii de compus pentru n.

Pentru a completa demonstraţia, vom arăta că nonprobele sunt conţinute toate într-un subgrup B propriu al lui $\mathbb{Z}_n^\star$. Din corolarul 33.16 avem $|B| \le |\mathbb{Z}_n^\star|/2$. Deoarece $|\mathbb{Z}_n^\star| \le n-1$, obţinem $|B| \le (n-1)/2$. De aceea, numărul de nonprobe este cel mult $(n-1)/2$, deci numărul de probe trebuie să fie cel puţin $(n-1)/2$.

Să arătăm modalitatea de obţinere a unui subgrup propriu B al lui $\mathbb{Z}_n^\star$ care conţine toate nonprobele. Împărţim demonstraţia în două cazuri.

Cazul 1: Există un $x \in \mathbb{Z}_n^\star$, astfel încât:

$$x^{n-1} \not\equiv 1 \pmod{n}. \tag{33.43}$$

Fie $B = \{b \in \mathbb{Z}_n^\star : b^{n-1} \equiv 1 \pmod{n}\}$. Întrucât B este o mulţime închisă în raport cu înmulţirea modulo n, rezultă că B este un subgrup al lui $\mathbb{Z}_n^\star$, conform teoremei 33.14. Să observăm că orice nonprobă aparţine lui B, deoarece o nonprobă a satisface $a^{n-1} \equiv 1 \pmod{n}$. Deoarece $x \in \mathbb{Z}_n^\star - B$, rezultă că B este un subgrup propriu al lui $\mathbb{Z}_n^\star$.

Cazul 2: Pentru orice $x \in \mathbb{Z}_n^\star$,

$$x^{n-1} \equiv 1 \pmod{n}. \tag{33.44}$$

În acest caz, n nu poate fi o putere a unui număr prim. Pentru a vedea de ce, fie $n = p^e$ unde p este un număr prim impar şi $e > 1$. Teorema 33.32 implică faptul că $\mathbb{Z}_n^\star$ conţine un element g, astfel încât $\mathrm{ord}_n(g) = |\mathbb{Z}_n^\star| = \phi(n) = (p-1)p^{e-1}$. Dar atunci ecuaţia (33.44) şi teorema logaritmului discret (teorema 33.33 pentru $y = 0$) implică faptul că $n - 1 \equiv 0 \pmod{\phi(n)}$, sau

$$(p-1)p^{e-1} | p^e - 1.$$

Această condiţie eşuează pentru $e > 1$, deoarece în caz contrar partea stângă ar fi divizibilă cu p dar partea dreaptă nu. Astfel, n nu este o putere a unui număr prim.

Întrucât n nu este o putere a unui număr prim, îl descompunem într-un produs $n_1 n_2$, unde n_1 şi n_2 sunt mai mari decât 1 şi relativi primi. (Pot exista diferite moduri de a realiza acest lucru şi nu are importanţă care anume a fost ales. De exemplu, dacă $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, atunci putem alege $n_1 = p_1^{e_1}$ şi $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Se definesc t şi u astfel încât $n - 1 = 2^t u$, unde $t \geq 1$ şi u este impar. Pentru orice $a \in \mathbb{Z}_n^\star$, considerăm şirul

$$\hat{a} = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle, \tag{33.45}$$

unde toate elementele sunt calculate modulo n. Deoarece $2^t | n - 1$, reprezentarea binară a lui $n-1$ se termină cu t zerouri, iar elementele lui $\hat{a}$ sunt ultimele $t+1$ valori ale lui d, calculate prin PROBA pe parcursul unui calcul al lui $a^{n-1} \bmod n$; ultimele t operaţii sunt ridicări la pătrat.

Determinăm un $j \in \{0, 1, \ldots, t\}$ astfel încât să existe un $v \in \mathbb{Z}_n^\star$ pentru care $v^{2^j u} \equiv -1 \pmod{n}$; j ar trebui să fie cât mai mare posibil. Un astfel de j există cu siguranţă deoarece u este impar: putem alege $v = -1$ şi $j = 0$. Fixăm pe v astfel încât condiţia dată să fie satisfăcută. Fie

$$B = \{x \in \mathbb{Z}_n^\star : x^{2^j u} \equiv \pm 1 \pmod{n}\}.$$

Întrucât mulţimea B este închisă în raport cu înmulţirea modulo n, ea este un subgrup al lui $\mathbb{Z}_n^\star$. De aceea, $|B|$ divide $|\mathbb{Z}_n^\star|$. Orice nonprobă trebuie să fie un element al mulţimii B, deoarece şirul (33.45) produs de o nonprobă trebuie să aibă toţi termenii 1 sau să conţină un element egal cu -1 pe primele j poziţii, datorită maximalităţii lui j.

Folosim acum existenţa lui v pentru a demonstra că există un $w \in \mathbb{Z}_n^\star - B$. Întrucât $v^{2^j u} \equiv -1 \pmod{n}$, avem $v^{2^j u} \equiv -1 \pmod{n_1}$ conform corolarului 33.29. Pe baza corolarului 33.28, există un w care satisface simultan ecuaţiile:

$$\begin{aligned} w &= v \pmod{n_1}, \\ w &= 1 \pmod{n_2}. \end{aligned}$$

De aceea,

$$\begin{aligned} w^{2^j u} &\equiv -1 \pmod{n_1}, \\ w^{2^j u} &\equiv 1 \pmod{n_2}. \end{aligned}$$

Împreună cu corolarul 33.29, aceste ecuaţii implică faptul că

$$w^{2^j u} \not\equiv \pm 1 \pmod{n}, \tag{33.46}$$

şi astfel $w \notin B$. Întrucât $v \in \mathbb{Z}_n^\star$, avem că $v \in \mathbb{Z}_{n_1}^\star$. Astfel, $w \in \mathbb{Z}_n^\star$ şi $w \in \mathbb{Z}_n^\star - B$. Deducem că B este un subgrup propriu al lui $\mathbb{Z}_n^\star$.

În ambele cazuri, se observă că numărul probelor pentru proprietatea de număr compus al lui n este cel puţin $(n-1)/2$. ∎

Teorema 33.39 Pentru orice întreg impar $n > 2$ şi orice întreg pozitiv s, probabilitatea ca MILLER-RABIN(n, s) să greşească este de cel mult 2^{-s}.

Demonstraţie. Utilizând teorema 33.38, observăm că, dacă n este compus, atunci fiecare execuţie a ciclului din liniile 1–4 descoperă o probă x pentru proprietatea de număr compus al lui n, cu o probabilitate de cel puţin $1/2$. MILLER-RABIN greşeşte numai dacă este aşa de nenorocoasă încât nu găseşte o probă pentru proprietatea de număr compus al lui n la nici una din cele s iteraţii ale ciclului principal. Probabilitatea unui astfel de şir de ratări este de cel mult 2^{-s}. ∎

Astfel, $s = 50$ ar trebui să fie suficient pentru aproape orice aplicaţie imaginabilă. Dacă încercăm să găsim numere prime mari aplicând MILLER-RABIN la întregi mari *aleşi aleator*, atunci se poate argumenta (deşi nu o vom face aici) că alegând o valoare mică pentru s (de exemplu 3) este puţin probabil să fim conduşi la rezultate eronate. Adică, pentru un număr întreg impar n compus, ales aleator, numărul mediu de nonprobe pentru proprietatea de număr compus a lui n este probabil mult mai mic decât $(n-1)/2$. Dacă întregul n nu este ales aleator, totuşi, cel mai bun rezultat de până acum (folosind o versiune îmbunătăţită a teoremei 33.39) este că numărul de nonprobe este cel mult $(n-1)/4$. Mai mult decât atât, există numere întregi n pentru care numărul de nonprobe este $(n-1)/4$.

Exerciţii

33.8-1 Demonstraţi că, dacă un întreg impar $n > 1$ nu este un număr prim sau o putere a unui număr prim, atunci există o rădăcină pătrată netrivială a lui 1 modulo n.

33.8-2 ⋆ Este posibil să se întărească puţin teorema lui Euler sub forma

$$a^{\lambda(n)} \equiv 1 \pmod{n} \text{ pentru orice } a \in \mathbb{Z}_n^\star,$$

unde $\lambda(n)$ se defineşte prin

$$\lambda(n) = \text{ cmmmc}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})). \tag{33.47}$$

Arătaţi că $\lambda(n) \mid \phi(n)$. Un număr compus n este un număr Carmichael dacă $\lambda(n) \mid n-1$. Cel mai mic număr Carmichael este $561 = 3 \cdot 11 \cdot 17$; aici $\lambda(n) = \text{cmmmc}(2, 10, 16) = 80$, care divide pe 560. Demonstraţi că numerele Carmichael trebuie să fie atât "libere de pătrate" (să nu fie divizibile prin pătratul nici unui număr prim) cât şi produsul a cel puţin trei numere prime. Din acest motiv, ele nu sunt prea răspândite.

33.8-3 Arătaţi că, dacă x este o rădăcină pătrată netrivială a lui 1 modulo n, atunci $\text{cmmdc}(x-1, n)$ şi $\text{cmmdc}(x+1, n)$ sunt divizori netriviali ai lui n.

33.9. Factorizarea întreagă

Presupunem că avem un întreg n pe care dorim să-l ***factorizăm***, adică, să-l descompunem într-un produs de numere prime. Testul de număr prim, din secţiunea precedentă, ne va spune că n este compus, dar nu ne va spune şi factorii lui n. Factorizarea unui număr întreg mare n pare a fi mult mai dificilă decât simpla determinare dacă n este prim sau compus. Cu supercalculatoarele de azi şi cu cei mai buni algoritmi curenţi factorizarea unui număr arbitrar cu 200 de cifre zecimale este irealizabilă.

Euristica rho a lui Pollard

Încercarea de a împărţi cu toate numerele întregi până la B garantează factorizarea completă a oricărui număr până la B^2. Cu aceeaşi cantitate de muncă, următoarea procedură va factoriza orice număr până la B^4 (exceptând cazul în care suntem nenorocoşi). Întrucât procedura este doar euristică, nici timpul ei de execuţie şi nici succesul ei nu sunt garantate, deşi este foarte eficientă în practică.

```
POLLARD-RHO(n)
 1: i ← 1
 2: x_1 ← RANDOM(0, n − 1)
 3: y ← x_1
 4: k ← 2
 5: cât timp ADEVĂRAT execută
 6:    i ← i + 1
 7:    x_i ← (x_{i−1}^2 − 1) mod n
 8:    d ← cmmdc(y − x_i, n)
 9:    dacă d ≠ 1 şi d ≠ n atunci
10:       scrie d
11:       dacă  i = k atunci
12:          y ← x_i
13:          k ← 2k
```

Procedura funcţionează după cum urmează: liniile 1–2 iniţializează pe i cu 1 şi pe x_1 cu o valoare aleasă aleator din $\mathbb{Z}_n$. Ciclul **cât timp**, care începe în linia 5, iterează la infinit căutând factorii lui n. În timpul fiecărei iteraţii a ciclului **cât timp**, relaţia de recurenţă:

$$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n \tag{33.48}$$

este utilizată în linia 7 pentru a produce valoarea următoare a lui x_i din şirul infinit

$$x_1, x_2, x_3, x_4, \ldots; \tag{33.49}$$

valoarea lui i este incrementată corespunzător în linia 6. Codul este scris utilizând variabilele x_i cu indici pentru claritate, dar programul funcţionează la fel dacă se elimină indicii deoarece doar valoarea cea mai recentă a lui x_i trebuie păstrată.

Programul salvează cea mai recent generată valoare x_i în variabila y. Mai exact, valorile care sunt salvate sunt cele care au indici puteri ale lui 2:

$$x_1, x_2, x_4, x_8, x_{16}, \ldots .$$

Linia 3 salvează valoarea x_1, iar linia 12 valoarea x_k ori de câte ori i este egal cu k. Variabila k se iniţializează cu 2 în linia 4, k este dublat în linia 13 ori de câte ori y este actualizat. De aceea, k urmează şirul 1, 2, 4, 8, ... şi defineşte, întotdeauna, indicele următoarei valori x_k de salvat în y. Liniile 8–10 încearcă să găsească un factor al lui n utilizând valoarea salvată a lui y şi valoarea curentă a lui x_i. Mai exact, linia 8 calculează cel mai mare divizor comun $d = \text{cmmdc}(y - x_i, n)$. Dacă d este un divizor netrivial al lui n (verificat în linia 9), atunci linia 10 îl scrie pe d.

Această procedură concepută pentru a găsi un factor poate părea, într-o oarecare măsură, misterioasă pentru început. Totuşi, să observăm că POLLARD-RHO nu afişează niciodată un răspuns incorect; orice număr pe care-l afişează este un divizor netrivial al lui n. Totuşi, POLLARD-RHO poate să nu afişeze nimic; nu există nici o garanţie că algoritmul va produce vreun rezultat. Vom vedea că există, totuşi, un motiv bun să ne aşteptăm ca POLLARD-RHO să afişeze un factor p al lui n după aproximativ $\sqrt{p}$ iteraţii ale ciclului **cât timp**. Astfel, dacă n este compus, ne putem aştepta ca această procedură să descopere destui divizori pentru a factoriza complet pe n după aproximativ $n^{1/4}$ actualizări, deoarece fiecare factor prim p al lui n, exceptându-l pe cel mai mare posibil, este mai mic decât $\sqrt{n}$.

Analizăm comportamentul acestei proceduri studiind cât de lung este un şir aleator, modulo n, pentru a repeta o valoare. Deoarece $\mathbb{Z}_n$ este finit şi fiecare valoare din şirul (33.49) depinde numai de valoarea precedentă, se repetă, eventual, şirul (33.49). O dată ce se ajunge la x_i, astfel încât $x_i = x_j$ pentru $j < i$, suntem într-un ciclu, deoarece $x_{i+1} = x_{j+1}$, $x_{i+2} = x_{j+2}$ şi aşa mai departe. Motivul pentru care acest procedeu se numeşte "euristica rho" este acela că, aşa cum arată figura 33.7, şirul $x_1, x_2, \ldots, x_{j-1}$ poate fi desemnat ca o "coadă" a lui rho, iar ciclul $x_j, x_{j+1}, \ldots, x_i$ drept "corpul" lui rho.

Să examinăm cât durează până când şirul x_i începe să se se repete. Cu puţine modificări, vom putea folosi mai târziu rezultatul obţinut.

În scopul acestei estimări, să presupunem că funcţia $(x^2 - 1) \bmod n$ se comportă ca o funcţie "aleatoare". Desigur, în realitate, ea nu este aleatoare, dar această presupunere produce rezultate compatibile cu comportamentul algoritmului POLLARD-RHO. Putem, apoi, considera că fiecare x_i a fost ales independent din $\mathbb{Z}_n$ potrivit unei distribuţii uniforme pe $\mathbb{Z}_n$. Prin analiza paradoxului zilei de naştere, din secţiunea 6.6.1, numărul mediu de paşi anteriori ciclării şirului este $\Theta(\sqrt{n})$.

Acum, urmează modificarea cerută. Fie p un factor netrivial al lui n, astfel încât cmmdc(p, n/p) = 1. De exemplu, dacă n are factorizarea $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, putem să-l luăm pe p chiar $p_1^{e_1}$. (Dacă $e_1 = 1$, atunci p este chiar cel mai mic factor prim al lui n, un exemplu bun de ţinut minte.) Şirul $\langle x_i \rangle$ induce un şir corespunzător $\langle x_i' \rangle$ modulo p, unde

$$x_i' = x_i \bmod p$$

pentru orice i. Mai mult decât atât, din teorema chineză a restului avem

$$x_{i+1}' = (x_i'^2 - 1) \bmod p \tag{33.50}$$

deoarece

$$(x \bmod n) \bmod p = x \bmod p,$$

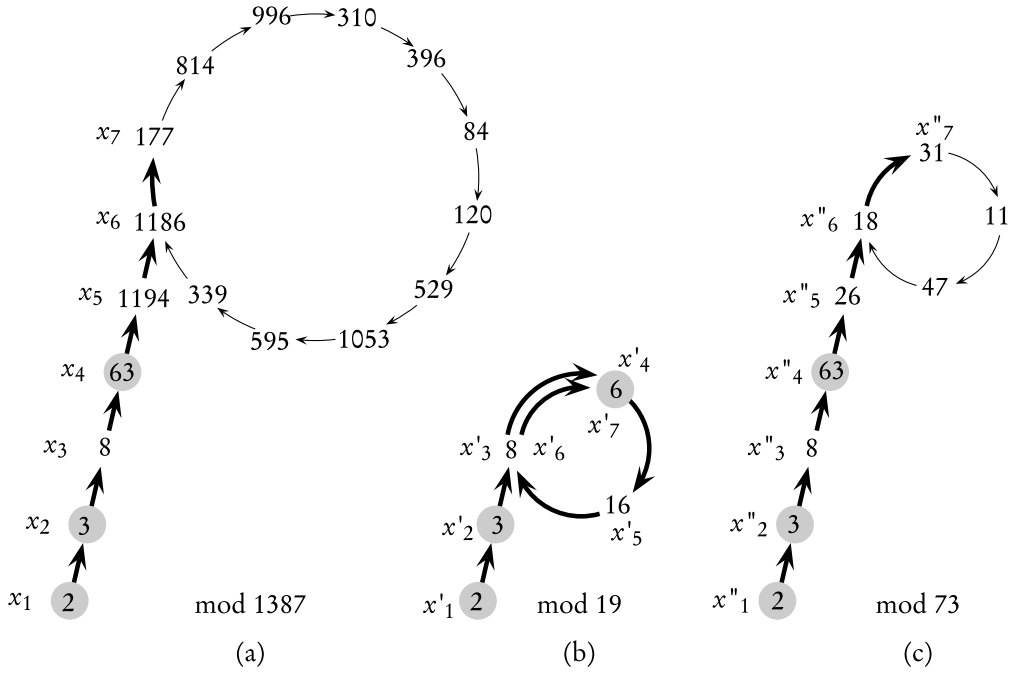


Figura 33.7 Euristica rho a lui Pollard. **(a)** Valorile produse prin recurenţa $x_{i+1} \leftarrow (x_i^2 - 1) \bmod 1387$, începând cu $x_1 = 2$. Factorizarea în numere prime a lui 1387 este $19 \cdot 73$. Săgeţile îngroşate indică paşii iteraţiei care sunt executaţi înainte ca factorul 19 să fie descoperit. Săgeţile subţiri indică spre valori neatinse în iteraţie pentru a ilustra forma de "rho". Valorile haşurate sunt valorile y memorate prin POLLARD-RHO. Factorul 19 este descoperit după ce este atins $x_7 = 177$, când este calculat $\text{cmmdc}(63 - 177, 1387) = 19$. Prima valoare x care va fi repetată este 1186, dar factorul 19 este descoperit înainte de a atinge această valoare. **(b)** Valorile produse prin aceeaşi recurenţă, modulo 19. Fiecare valoare x_i, dată în partea (a), este echivalentă modulo 19 cu valoarea x'_i indicată aici. De exemplu, atât $x_4 = 63$, cât şi $x_7 = 177$ sunt echivalente cu 6 modulo 19. **(c)** Valorile produse prin aceeaşi recurenţă, modulo 73. Fiecare valoare x_i, prezentată în partea (a), este echivalentă, modulo 73, cu valoarea x''_i prezentată aici. Din teorema chineză a restului, rezultă că fiecare nod din partea (a) corespunde unei perechi de noduri, unul din partea (b) şi unul din partea (c).

pe baza exerciţiului 33.1-6.

Raţionând analog, găsim că numărul mediu de paşi, înainte ca şirul $\langle x'_i \rangle$ să se repete, este $\Theta(\sqrt{p})$. Dacă p este mic în comparaţie cu n, şirul $\langle x'_i \rangle$ se poate repeta mult mai rapid dacât şirul $\langle x_i \rangle$. Într-adevăr, şirul $\langle x'_i \rangle$ se repetă de îndată ce două elemente ale şirului $\langle x_i \rangle$ sunt echivalente numai modulo p, şi nu echivalente modulo n. Pentru o ilustrare, vezi figura 33.7, părţile (b) şi (c).

Fie t indicele primei valori repetate în şirul $\langle x'_i \rangle$ şi fie $u > 0$ lungimea ciclului care a fost produs în acest fel. Adică, t şi $u > 0$ sunt cele mai mici valori, astfel încât $x'_{t+i} = x'_{t+u+i}$ pentru orice $i \geq 0$. Prin argumentele de mai sus, valorile medii ale lui t şi u sunt $\Theta(\sqrt{p})$. Să observăm că, dacă $x'_{t+i} = x'_{t+u+i}$, atunci $p \mid (x_{t+u+i} - x_{t+i})$. Astfel, $\text{cmmdc}(x_{t+u+i} - x_{t+i}, n) > 1$.

De aceea, o dată ce POLLARD-RHO a salvat în y orice valoare x_k, astfel încât $k \geq t$, atunci

y mod p este totdeauna în ciclul modulo p. (Dacă o valoare nouă este salvată în y, acea valoare este, de asemenea, în ciclul modulo p.) În final, k primeşte o valoare care este mai mare decât u şi apoi procedura face un ciclu întreg în jurul ciclului modulo p fără a schimba valoarea lui y. Apoi, este găsit un factor al lui n când x_i se execută cu "valoarea memorată" în prealabil în y, modulo p, adică atunci când $x_i \equiv y \pmod{p}$.

Probabil că factorul găsit este factorul p, deşi se poate, în mod ocazional, să se descopere un multiplu al lui p. Deoarece valorile medii, atât ale lui t, cât şi ale lui u sunt $\Theta(\sqrt{p})$, numărul mediu de paşi necesari pentru a determina factorul p este $\Theta(\sqrt{p})$.

Există două motive pentru care acest algoritm nu se execută aşa cum este de aşteptat. Întâi, analiza euristică a timpului de execuţie nu este riguroasă şi este posibil ca ciclul de valori, modulo p, să fie mai mare decât $\sqrt{p}$. În acest caz, algoritmul se execută corect, dar mult mai încet decât s-a sperat. În practică, aceasta se pare că nu este o problemă. În al doilea rând, divizorii lui n produşi de acest algoritm ar putea, totdeauna, să fie unul din factorii triviali 1 sau n. De exemplu, presupunem că $n = pq$, unde p şi q sunt numere prime. Se poate întâmpla ca valorile lui t şi u, pentru p, să fie identice cu valorile lui t şi u pentru q şi astfel factorul p este totdeauna găsit în aceeaşi operaţie cmmdc care descoperă factorul q. Deoarece ambii factori sunt descoperiţi în acelaşi timp, este descoperită factorizarea trivială $pq = n$, care este inutilă. Din nou, aceasta nu pare a fi o problemă reală în practică. Dacă este necesar, se poate reîncepe euristica, cu o recurenţă diferită, de forma $x_{i+1} \leftarrow (x_i^2 - c) \bmod n$. (Valorile $c = 0$ şi $c = 2$ ar trebui să fie eliminate pentru motive nejustificate aici, dar alte valori sunt bune.)

Desigur, această analiză este euristică şi nu este riguroasă, deoarece recurenţa nu este în realitate "aleatoare". Mai mult decât atât, procedura se execută bine în practică şi se pare că este eficientă aşa cum indică această analiză euristică. Este o metodă recomandată pentru a găsi factori primi mici ai unui număr mare. Pentru a factoriza complet un număr compus n, reprezentat pe β biţi, este nevoie numai să găsim toţi factorii primi mai mici decât $\lfloor n^{1/2} \rfloor$ şi ne putem aştepta ca POLLARD-RHO să necesite cel mult $n^{1/4} = 2^{\beta/4}$ operaţii aritmetice şi cel mult $n^{1/4}\beta^3 = 2^{\beta/4}\beta^3$ operaţii pe biţi. Abilitatea lui POLLARD-RHO de a găsi un factor mic p al lui n cu un număr mediu $\Theta(\sqrt{p})$ de operaţii aritmetice este deseori cea mai atractivă caracteristică.

Exerciţii

33.9-1 Conform execuţiei acţiunilor vizualizate în figura 33.7 (a), când se tipăreşte factorul 73 al lui 1387 în procedura POLLARD-RHO?

33.9-2 Presupunem că se dă o funcţie $f : \mathbb{Z}_n \to \mathbb{Z}_n$ şi o valoare iniţială $x_0 \in \mathbb{Z}_n$. Definim $x_i = f(x_{i-1})$ pentru $i = 1, 2, \ldots$. Fie t şi $u > 0$ cele mai mici valori astfel încât $x_{t+i} = x_{t+u+i}$ pentru $i = 0, 1, \ldots$. În terminologia algoritmului rho a lui Pollard, t este lungimea cozii şi u este lungimea ciclului lui rho. Elaboraţi un algoritm eficient pentru a determina exact pe t şi pe u şi analizaţi timpul de execuţie.

33.9-3 Câţi paşi va necesita POLLARD-RHO pentru a descoperi un factor de forma p^e, unde p este prim şi $e > 1$?

33.9-4 $\star$ Un dezavantaj al lui POLLARD-RHO, aşa cum s-a scris, este că cere calculul lui cmmdc pentru fiecare pas al recurenţei. S-a sugerat să grupăm calculul cmmdc acumulând produsul diferiţilor x_i într-o linie şi luând, apoi, cmmdc pentru acest produs, cu valoarea salvată în y. Descrieţi cum s-ar putea implementa această idee, de ce funcţionează şi ce dimensiune ar trebui

aleasă pentru a fi cea mai eficientă atunci când se lucrează asupra unui număr n reprezentat pe β biţi.

Probleme

33-1 Algoritm cmmdc binar

Pe cele mai multe calculatoare, operaţiile de scădere, de testare a parităţii (impar sau par) unui număr întreg binar şi înjumătăţire pot fi realizate mai rapid decât calculul resturilor. Această problemă studiază ***algoritmul cmmdc binar***, care elimină calculul resturilor utilizat în algoritmul lui Euclid.

a. Demonstraţi că, dacă a şi b sunt pari, atunci $\text{cmmdc}(a, b) = 2\,\text{cmmdc}(a/2, b/2)$.

b. Demonstraţi că, dacă a este impar şi b par, atunci $\text{cmmdc}(a, b) = \text{cmmdc}(a, b/2)$.

c. Demonstraţi că, dacă a şi b sunt impari, atunci $\text{cmmdc}(a, b) = \text{cmmdc}((a - b)/2, b)$.

d. Proiectaţi un algoritm cmmdc binar eficient pentru numerele întregi a şi b, unde $a \geq b$, care să se execute într-un timp $O(\lg(\max(a, b))$. Se presupune că fiecare scădere, test de paritate şi înjumătăţire se execută într-o unitate de timp.

33-2 Analiza operaţiilor pe biţi în algoritmul lui Euclid

a. Arătaţi că utilizarea algoritmului banal cu "creion şi hârtie" pentru împărţirea a două numere întregi mari a şi b, obţinându-se câtul q şi restul r, necesită $O((1 + \lg q) \lg b)$ operaţii pe biţi.

b. Se defineşte $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Arătaţi că numărul operaţiilor pe biţi executate de algoritmul EUCLID, pentru a reduce problema calculului lui $\text{cmmdc}(a, b)$ la cea a lui $\text{cmmdc}(b, a \bmod b)$, este cel mult $c(\mu(a, b) - \mu(b, a \bmod b))$ pentru o constantă $c > 0$ suficient de mare.

c. Arătaţi că algoritmul EUCLID(a, b) necesită, $O(\mu(a, b))$ operaţii pe biţi în general, şi $O(\beta^2)$ operaţii pe biţi dacă algoritmul are ca intrare două numere întregi reprezentate pe β biţi.

33-3 Trei algoritmi pentru numerele Fibonacci

Această problemă compară eficienţa a trei metode pentru calculul celui de al n-lea număr al lui Fibonacci F_n, pentru un n dat. Costul pentru adunare, scădere sau înmulţire a două numere este $O(1)$, independent de dimensiunea numerelor.

a. Arătaţi că timpul de execuţie al metodei recursive directe pentru calculul lui F_n, bazat pe recurenţa (2.13), este exponenţial în raport cu n.

b. Arătaţi cum se poate calcula F_n într-un timp de ordinul lui $O(n)$ utilizând memorarea.

c. Arătaţi cum se poate calcula F_n într-un timp $O(\lg n)$ utilizând numai adunări şi înmulţiri de întregi. (*Indicaţie:* se consideră matricea

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

şi puterile ei).

d. Se presupune acum că, adunarea a două numere având reprezentarea binară pe β biţi necesită un timp $\Theta(\beta)$ şi că, înmulţirea a două numere având reprezentarea binară pe β biţi necesită timpul $\Theta(\beta^2)$. Care este timpul de execuţie al acestor trei metode considerând această măsură a costului în timp, mult mai rezonabilă pentru operaţiile aritmetice elementare?

33-4 Reziduuri pătratice

Fie p un număr prim impar. Un număr $a \in \mathbb{Z}_p^\star$ este un ***reziduu pătratic*** dacă ecuaţia $x^2 = a \pmod p$ are o soluţie pentru necunoscuta x.

a. Arătaţi că există exact $(p-1)/2$ reziduuri pătratice modulo p.

b. Dacă p este prim, definim ***simbolul lui Legendre*** $\left(\frac{a}{p}\right)$ pentru $a \in \mathbb{Z}_p^\star$, ca fiind 1, dacă a este un reziduu pătratic modulo p, şi -1 altfel. Arătaţi că, dacă $a \in \mathbb{Z}_p^\star$, atunci

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p.$$

Elaboraţi un algoritm eficient pentru a determina dacă un număr dat a este sau nu un reziduu pătratic modulo p. Să se analizeze eficienţa algoritmului.

c. Demonstraţi că, dacă p este prim de forma $4k+3$ şi a este un reziduu pătratic în $\mathbb{Z}_p^\star$, atunci $a^{k+1} \bmod p$ este o rădăcină pătrată a lui a, modulo p. Cât timp este necesar pentru a găsi rădăcina pătrată dintr-un reziduu pătratic a modulo p?

d. Descrieţi un algoritm de randomizare eficient pentru a găsi un reziduu nepătratic, modulo un număr prim p arbitrar. Câte operaţii aritmetice necesită, în medie, algoritmul?

Note bibliografice

Niven şi Zuckerman [151] furnizează o introducere excelentă în teoria elementară a numerelor. Lucrarea lui Knuth [122] conţine o discuţie bună despre algoritmul de găsire a celui mai mare divizor comun, precum şi despre alţi algoritmi din teoria numerelor. Riesel [168] şi Bach [16] furnizează studii mai recente asupra teoriei calculului numeric. Dixon [56] dă o privire generală asupra factorizării şi testării proprietăţii de număr prim. Lucrările conferinţei editate de Pomerance [159] conţin diferite articole interesante de ordin general.

Knuth [122] discută originea algoritmului lui Euclid. El a apărut în cartea 7, propoziţiile 1 şi 2 din lucrarea *Elementele* a matematicianului grec Euclid, care a fost scrisă în jurul anului 300 î.Hr. Descrierea lui Euclid ar putea deriva din algoritmul dat de Eudoxus în jurul anului 375 î.Hr. Algoritmul lui Euclid are onoarea de a fi cel mai vechi algoritm netrivial şi cu acesta rivalizează numai algoritmul ţăranului rus pentru înmulţire – capitolul 29, care a fost cunoscută în Egiptul antic.

Knuth atribuie un caz particular al teoremei chinezeşti a restului matematicianului chinez Sun-Tsŭ care a trăit, aproximativ, între 200 î.Hr. şi 200 d.Hr. – data este foarte inexactă. Acelaşi

caz particular a fost dat de matematicianul grec Nichomachus în jurul anului 100 d.Hr. Acesta a fost generalizat de Chhin Chiu-Shao în 1247. Teorema chineză a restului a fost, în final, enunţată şi demonstrată complet, în generalitatea sa, de către L. Euler în 1734.

Algoritmul aleator de test al primalităţii prezentat aici se datorează lui Miller [147] şi Rabin [166]; este cel mai rapid algoritm aleator cunoscut, care verifică proprietatea de număr prim, abstracţie făcând de factori constanţi. Demonstraţia teoremei 33.39 este o uşoară adaptare a celei sugerate de Bach [15]. O demonstraţie a unui rezultat mai puternic pentru MILLER-RABIN a fost dată de Monier [148, 149]. Randomizarea s-a dovedit a fi necesară pentru a obţine un algoritm care verifică proprietatea de primalitate, într-un timp polinomial. Cel mai rapid algoritm care verifică proprietatea de primalitate, demonstrat este versiunea Cohen-Lenstra [45] a testului de primalitate al lui Adleman, Pomerance şi Rumely [3]. Testul care verifică dacă un număr n de lungime $\lceil \lg(n+1) \rceil$ este prim, se execută în timpul $(\lg n)^{O(\lg \lg \lg n)}$, ceea ce este chiar uşor superpolinomial.

Problema găsirii "aleatoare" a numerelor prime mari este discutată frumos într-un articol al lui Beauchemin, Brassard, Crépeau, Goutier şi Pomerance [20].

Conceptul de criptosistem cu cheie publică se datorează lui Diffie şi Hellman [54]. Criptosistemul RSA a fost propus în 1977 de Rivest, Shamir şi Adleman [169]. De atunci, domeniul criptografiei a cunoscut o dezvoltare deosebită. În particular, au fost dezvoltate tehnici noi pentru a demonstra siguranţa criptosistemelor. De exemplu, Goldwasser şi Micali [86] au arătat că randomizarea poate fi un instrument eficient în proiectarea schemelor de criptare sigure cu cheie publică. Pentru schemele de semnături, Goldwasser, Micali şi Rivest [88] prezintă o schemă de semnătură digitală pentru care fiecare tip de falsificare posibil de imaginat este, probabil, tot atât de dificil ca factorizarea. Recent, Goldwasser, Micali şi Rackoff [87] au introdus o clasă de scheme de criptare de "cunoştinţe nule" pentru care se poate demonstra (în anumite presupuneri rezonabile) că nici o parte nu învaţă mai mult decât i s-a propus să înveţe dintr-o comunicaţie.

Euristica rho pentru factorizarea întreagă a fost inventată de Pollard [156]. Versiunea prezentată aici este o variantă propusă de Brent [35].

Cei mai buni algoritmi de factorizare a numerelor mari au un timp de execuţie care creşte exponenţial cu rădăcina pătrată din lungimea numărului n de factorizat. Algoritmul de factorizare prin rafinare pătrratică datorat lui Pomerance [158] este, probabil, cel mai eficient algoritm de acest fel pentru intrări mari. Deşi este dificil să se dea o analiză riguroasă a acestui algoritm, în condiţii rezonabile, putem deduce o estimare a timpului de execuţie prin $L(n)^{1+o(1)}$, unde $L(n) = e^{\sqrt{\ln n \ln \ln n}}$. Metoda curbei eliptice, datorată lui Lenstra [137], poate fi mai eficientă pentru anumite date de intrare decât metoda prin rafinare pătratică, deoarece, la fel ca metoda rho a lui Pollard, ea poate găsi un factor prim mic destul de repede. Cu această metodă, timpul pentru determinarea lui p este estimat la $L(p)^{\sqrt{2}+o(1)}$.

34 Potrivirea şirurilor

Determinarea tuturor aparţiilor unui model, într-un text, este o problemă frecvent întâlnită la editoarele de texte. De obicei, textul este un document în editare şi modelul căutat este un anumit cuvânt, dat de utilizator. Algoritmi eficienţi pentru această problemă pot ajuta la îmbunătăţirea performanţelor editoarelor de texte. Algoritmi de potrivire a şirurilor sunt utilizaţi, de asemenea, în căutarea de modele particulare în secvenţe ADN.

În continuare, vom prezenta un model formal pentru ***problema potrivirii şirurilor***. Presupunem că textul este un vector $T[1..n]$ de lungime n şi că modelul căutat este un vector $P[1..m]$ de lungime $m \leq n$. În plus, presupunem că elementele din P şi T au caractere dintr-un alfabet finit Σ. De exemplu putem avea $\Sigma = \{0, 1\}$ sau $\Sigma = \{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$. Tablourile de caractere P şi T sunt, de obicei, numite ***şiruri*** de caractere.

Spunem că modelul P ***apare cu deplasament s*** în textul T (sau, echivalent, că modelul P ***apare începând cu poziţia s+1*** în textul T) dacă $0 \leq s \leq n - m$ şi $T[s + 1..s + m] = P[1..m]$ (adică, dacă $T[s + j] = P[j]$, pentru $1 \leq j \leq m$). Dacă P apare cu deplasament s în T, atunci numim s un ***deplasament corect***; altfel, numim s un ***deplasament incorect***. Problema potrivirii şirurilor este problema determinării tuturor deplasamentelor corecte cu care un model dat P apare într-un text dat T. Figura 34.1 ilustrează aceste definiţii.

Capitolul este organizat după cum urmează. În secţiunea 34.1, prezentăm algoritmul naiv forţă-brută pentru problema potrivirii şirurilor, care are, în cel mai defavorabil caz, timpul de execuţie $O((n - m + 1)m)$. În secţiunea 34.2, se prezintă un algoritm interesant de potrivire a şirurilor, datorat lui Rabin şi Karp. Acest algoritm are, în cel mai defavorabil caz, timpul de execuţie $O((n - m + 1)m)$, dar media de funcţionare este mult mai bună. De asemenea, se poate generaliza uşor pentru alte probleme de potrivire de modele. În continuare, în secţiunea 34.3, se descrie un algortim de potrivire a şirurilor care începe cu construirea unui automat finit, special proiectat să caute într-un text potriviri pentru un model dat P. Acest algoritm are timpul de execuţie $O(n + m|\Sigma|)$. Algoritmul Knuth-Morris-Pratt (sau KMP), similar, dar mult mai ingenios, este prezentat în secţiunea 34.4; algoritmul KMP se execută în timpul $O(n + m)$. În final, în secţiunea 34.5, se descrie un algortim datorat lui Boyer şi Moore care este adesea preferat în practică, deşi timpul său de execuţie, în cel mai defavorabil caz (ca şi la algoritmul Rabin-Karp), nu este mai bun decât cel al algoritmultui naiv pentru potrivirea şirurilor.

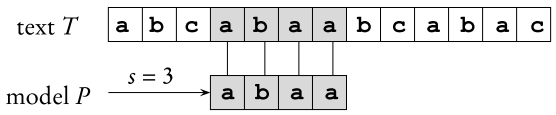


Figura 34.1 Problema potrivirii şirurilor. Scopul este determinarea tuturor aparţiilor modelului $P =$ `abaa` în textul $T =$ `abcabaabcabac`. Modelul apare în text o singură dată, cu deplasamentul $s = 3$. Deplasamentul $s = 3$ este un deplasament corect. Fiecare caracter din model este unit printr-o linie verticală cu caracterul identic regăsit în text şi toate caracterele regăsite sunt haşurate.

Notaţii şi terminologie

Vom nota cu Σ^* (citim "sigma stelat") mulţimea tuturor şirurilor de lungime finită formate, folosind caractere din alfabetul Σ. În acest capitol, considerăm numai şiruri de lungime finită. Şirul de lungime zero ***şirul vid***, notat cu ε, aparţine de asemenea lui Σ^*. Lungimea unui şir x este notată cu $|x|$. ***Concatenarea*** a două şiruri x şi y, pe care o notăm cu xy, are lungimea $|x| + |y|$ şi conţine caracterele din x urmate de caracterele din y.

Spunem că un şir w este un ***prefix*** al unui şir x, notat $w \sqsubset x$, dacă $x = wy$ pentru orice şir $y \in \Sigma^*$. De notat că, dacă $w \sqsubset x$, atunci $|w| \leq |x|$. În mod analog, spunem că şirul w este un ***sufix*** al unui şir x, notat $w \sqsupset x$, dacă $x = yw$ pentru orice $y \in \Sigma^*$. Din $w \sqsupset x$ rezultă că $|w| \leq |x|$. Şirul vid ε este atât sufix cât şi prefix pentru orice şir. De exemplu, avem `ab` $\sqsubset$ `abcca` şi `cca` $\sqsupset$ `abcca`. Este folositor să observăm că, pentru oricare şiruri x şi y şi orice caracter a, avem $x \sqsupset y$, dacă, şi numai dacă, $xa \sqsupset ya$. De asemenea, observăm că $\sqsubset$ şi $\sqsupset$ sunt relaţii tranzitive. Următoarea lemă va fi folositoare mai târziu.

Lema 34.1 (Lema de suprapunere a sufixului) Presupunem că x, y şi z sunt şiruri, astfel încât $x \sqsupset z$ şi $y \sqsupset z$. Dacă $|x| \leq |y|$, atunci $x \sqsupset y$. Dacă $|x| \geq |y|$, atunci $y \sqsupset x$. Dacă $|x| = |y|$, atunci $x = y$.

Demonstraţie. Urmăriţi figura 34.2 pentru o demonstratie grafică. ∎

Pentru prescurtarea scrierii, vom nota prefixul k-caracter $P[1..k]$, din modelul $P[1..m]$ prin P_k. Deci, $P_0 = \varepsilon$ şi $P_m = P = P[1..m]$. În mod analog, notăm prefixul k-caracter al textului T cu T_k. Folosind aceste notaţii, putem considera că problema potrivirii şirurilor este problema determinării tuturor deplasamentelor s în domeniul $0 \leq s \leq n - m$, astfel încât $P \sqsupset T_{s+m}$.

În pseudocod, vom permite ca operaţia de verificare a egalităţii a două şiruri să fie considerată o operaţie elementară. Dacă şirurile sunt comparate de la stânga la dreapta şi compararea se opreşte când este descoperită o greşeală, presupunem că timpul necesar pentru un astfel de test depinde, după o funcţie liniară, de numărul caracterelor potrivite descoperite. Mai precis, se presupune că testul "$x = y$" durează $\Theta(t+1)$, unde t este lungimea celui mai lung şir z, astfel încât $z \sqsubset x$ şi $z \sqsubset y$.

34.1. Algoritmul naiv pentru potrivirea şirurilor

Algoritmul naiv găseşte toate deplasările corecte folosind un ciclu care satisface condiţia $P[1..m] = T[s+1..s+m]$ pentru fiecare dintre cele $n - m + 1$ valori posibile ale lui s.

```
POTRIVIRE-NAIVĂ-A-ŞIRURILOR(T, P)
1: n ← lungime[T]
2: m ← lungime[P]
3: pentru s ← 0, n − m execută
4:    dacă P[1..m]=T[s + 1..s + m] atunci
5:       tipăreşte "Modelul apare cu deplasamentul" s
```

Procedura naivă pentru potrivirea şirurilor poate fi interpretată grafic ca o deplasare peste text a unui "şablon" conţinând modelul, notând deplasamentele la care toate caracterele din

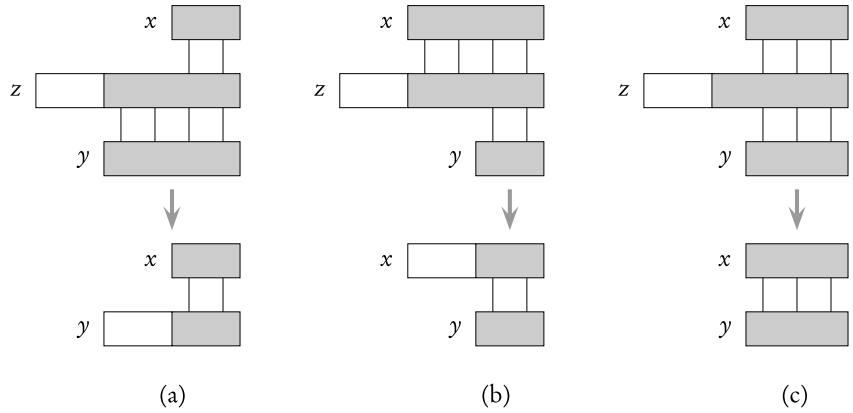


Figura 34.2 O demonstraţie grafică pentru lema 34.1. Presupunem că $x \sqsupset z$ şi $y \sqsupset z$. Cele trei cadre ale figurii ilustrează cele trei cazuri ale lemei. Liniile verticale unesc regiunile potrivite (haşurate) ale şirurilor. **(a)** Dacă $|x| \leq |y|$, atunci $x \sqsupset y$. **(b)** Dacă $|x| \geq |y|$, atunci $y \sqsupset x$. **(c)** Dacă $|x| = |y|$, atunci $x = y$.

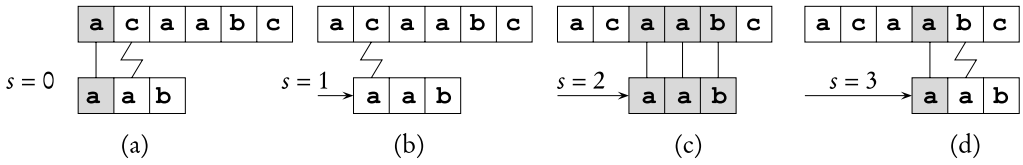


Figura 34.3 Funcţionarea algoritmului naiv pentru potrivirea şirurilor pentru modelul $P = \texttt{aab}$ şi textul $T = \texttt{acaabc}$. Ne putem imagina modelul P ca pe un "şablon" pe care îl vom suprapune peste text. Cadrele **(a)-(d)** prezintă patru poziţionări succesive încercate de algoritmul naiv pentru potrivire. În fiecare cadru, liniile verticale unesc regiunile corespunzătoare care se potrivesc (haşurate), iar o linie în zig-zag uneşte primul caracter găsit care nu se potriveşte, dacă acesta există. În cadrul (c) se observă o potrivire a modelului P, la deplasamentul $s = 2$.

şablon sunt egale cu caracterele corespondente din text, aşa cum se poate vedea în figura 34.3. În ciclul **pentru**, care începe în linia 3, sunt considerate explicit toate deplasamentele posibile. În linia 4, testul determină dacă deplasamentul curent este sau nu corect; acest test implică un ciclu implicit pentru verificarea corespondenţei poziţiilor caracterelor până când toate poziţiile se potrivesc sau până când este întâlnită prima nepotrivire. Linia 5 tipăreşte toate deplasamentele corecte s.

Procedura POTRIVIRE-NAIVĂ-A-ŞIRURILOR are, în cel mai defavorabil caz, timpul de execuţie $\Theta((n-m+1)m)$. De exemplu, considerăm textul $\texttt{a}^n$ (un şir format din n caractere a) şi modelul $\texttt{a}^m$. Pentru fiecare dintre cele $n - m + 1$ valori posibile ale deplasamentului s, ciclul implicit din linia 4 de comparare a caracterelor corespondente trebuie executat de m ori pentru a valida un deplasament. Timpul de execuţie, în cel mai defavorabil caz, este atunci $\Theta((n - m + 1)m)$, care este $\Theta(n^2)$ dacă $m = \lfloor n/2 \rfloor$.

După cum vom vedea, POTRIVIRE-NAIVĂ-A-ŞIRURILOR nu este procedura optimă pentru această problemă. Într-adevăr, în acest capitol, vom prezenta un algoritm cu timpul de execuţie, în cel mai defavorabil caz, $O(n + m)$. Algoritmul naiv este ineficient deoarece informaţia asupra

textului, obţinută pentru o valoare a lui s, este total ignorată la tratarea altor valori ale lui s. O astfel de informaţie poate fi foarte folositoare. De exemplu, dacă $P = \texttt{aaab}$ şi găsim că $s = 0$ este un deplasament corect, atunci nici unul dintre deplasamentele 1, 2 sau 3 nu este corect, deoarece $P[4] = \texttt{b}$. În secţiunile următoare vom examina câteva moduri pentru utilizarea acestui tip de informaţie.

Exerciţii

34.1-1 Arătaţi comparaţiile făcute de algoritmul naiv pentru potrivirea şirurilor în textul $T = \texttt{000010001010001}$ pentru modelul $P = \texttt{0001}$.

34.1-2 Arătaţi că algoritmul naiv pentru potrivirea şirurilor găseşte *prima* apariţie a unui model într-un text, în cel mai defavorabil caz, într-un timp $\Theta((n-m+1)(m-1))$.

34.1-3 Presupunem că, în modelul P, toate caracterele sunt diferite. Arătaţi cum poate algoritmul POTRIVIRE-NAIVĂ-A-ŞIRURILOR să se execute, pentru un text T de n caractere, în timpul $O(n)$.

34.1-4 Presupunem că modelul P şi textul P sunt şiruri obţinute *aleator*, de lungimi m şi respectiv, n, dintr-un alfabet $\Sigma_d = \{0, 1, ..., d-1\}$, unde $d \geq 2$. Arătaţi că numărul *mediu* de comparaţii caracater cu caracter executate de ciclul implicit din linia 4 a algoritmului naiv este

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1).$$

(Se presupune că algoritmul naiv termină compararea caracterelor, pentru un deplasament dat, o dată ce este găsită o nepotrivire sau modelul întreg este recunoscut.) Astfel, pentru şiruri alese aleator, algoritmul naiv este chiar eficient.

34.1-5 Presupunem că permitem modelului P să conţină apariţii ale unui ***caracter de discontinuitate*** $\diamondsuit$. Acesta se potriveşte cu orice şir *arbitrar* de caractere (chiar şi de lungime zero). De exemplu, modelul $\texttt{ab}\diamondsuit\texttt{ba}\diamondsuit\texttt{c}$ apare în textul `cabccbacbacab` astfel:

$$\texttt{c}\underbrace{\texttt{ab}}_{\texttt{ab}}\underbrace{\texttt{cc}}_{\diamondsuit}\underbrace{\texttt{ba}}_{\texttt{ba}}\underbrace{\texttt{cba}}_{\diamondsuit}\underbrace{\texttt{c}}_{\texttt{c}}\texttt{ab}$$

şi

$$\texttt{c}\underbrace{\texttt{ab}}_{\texttt{ab}}\underbrace{\texttt{ccbac}}_{\diamondsuit}\underbrace{\texttt{ba}}_{\texttt{ba}}\underbrace{}_{\diamondsuit}\underbrace{\texttt{c}}_{\texttt{c}}\texttt{ab}.$$

Caracterul de discontinuitate poate să apară ori de câte ori dorim în model, dar se presupune că nu apare în text. Daţi un algoritm polinomial pentru a determina dacă un astfel de model P apare într-un text dat T, şi analizaţi timpul de execuţie pentru acest algoritm.

34.2. Algoritmul Rabin-Karp

Rabin şi Karp au propus un algoritm pentru potrivirea şirurilor care se comportă bine în practică şi care, de asemenea, se poate generaliza la alţi algoritmi pentru probleme asemănătoare, cum ar fi potrivirea modelelor bidimensionale. Timpul de execuţie, în cel mai defavorabil caz, pentru algoritmul Rabin-Karp este $O((n-m+1)m)$, dar are un timp mediu de execuţie bun.

Acest algortim foloseşte noţiuni din teoria numerelor elementare, cum ar fi egalitatea a două numere modulo un al treilea număr. Pentru o definiţie exactă, se poate revedea secţiunea 33.1.

Pentru prezentarea propusă, presupunem că $\Sigma = \{0,1,2,\ldots,9\}$, deci fiecare caracter este o cifră zecimală. (În general, putem presupune că fiecare caracter este o cifră în baza d, unde $d = |\Sigma|$.) Putem, atunci, privi un şir de k caractere consecutive ca reprezentând un număr zecimal de lungime k. Astfel, şirul de caractere `31415` corespunde numărului zecimal `31,415`. Dat fiind faptul că datele de intrare pot fi interpretate pe de o parte ca simboluri grafice, pe de alta ca cifre, în continuare, le vom scrie cu fonturi obişnuite.

Fiind dat modelul $P[1..m]$, notăm cu p valoarea sa zecimală corespunzătoare. Într-o manieră similară, fiind dat textul $T[1..n]$, notăm cu t_s valoarea zecimală a subşirului $T[s+1..s+m]$ de lungime m, pentru $s = 0, 1, ..., n-m$. Desigur, $t_s = p$ dacă, şi numai dacă, $T[s+1..s+m] = P[1..m]$; astfel, s este un deplasament corect dacă, şi numai dacă, $t_s = p$. Dacă putem calcula p în timpul $O(m)$ şi toate valorile t_i în timpul total $O(n)$, atunci putem determina toate deplasamentele corecte s în timpul $O(n)$ prin compararea lui p cu fiecare t_s. (Pentru moment, nu ne îngrijorează faptul că p şi t_s ar putea fi numere foarte mari.)

Putem calcula p în timpul $O(m)$ folosind schema lui Horner (vezi secţiunea 32.1):

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots)).$$

Valoarea t_0 poate fi calculată, în mod analog, din $T[1..m]$ în timpul $O(m)$.

Pentru calcularea valorilor rămase $t_1, t_2, ..., t_{n-m}$, în timpul $O(n-m)$, este suficient să observăm că t_{s+1} poate fi calculat din t_s într-un timp constant, deoarece:

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]. \tag{34.1}$$

De exemplu, fie $m = 5$ şi $t_s = 31415$. Dacă dorim să ştergem cifra cea mai semnificativă $T[s+1] = 3$ şi să aducem, pe poziţia cea mai puţin semnificativă,atunci o nouă cifră (presupunem că aceasta este $T[s+5+1] = 2$) pentru a obţine:

$$t_{s+1} = 10(31415 - 10000 \cdot 3) + 2 = 14152.$$

Scăzând $10^{m-1}T[s+1]$, se şterge cifra cea mai semnificativă din t_s; înmulţind rezultatul cu 10, se deplasează numărul spre stânga cu o poziţie şi adunând la aceasta cifra $T[s+m+1]$, o aduce pe poziţia cea mai puţin semnificativă. Dacă, în prealabil, calculăm constanta 10^{m-1} (ceea ce se poate realiza în timpul $O(\lg m)$ folosind tehnicile din secţiunea 33.6, deşi pentru această aplicaţie este adecvată o metodă simplă $O(m)$), atunci fiecare execuţie a ecuaţiei (34.1) necesită un număr constant de operaţii aritmetice. Astfel, p şi $t_0, t_1, ..., t_{n-m}$ pot fi calculate în timpul $O(n+m)$ şi putem determina toate apariţiile modelului $P[1..m]$ în textul $T[1..n]$ într-un timp $O(n+m)$.

Singurul inconvenient la această procedură este că p şi t_s pot fi prea mari ca să se poată lucra, convenabil, cu ele. Dacă P conţine m caractere, atunci presupunerea că fiecare operaţie aritmetică asupra lui p (care are o lungime de m cifre) durează "un timp constant" nu este

rezonabilă. Din fericire, există o rezolvare simplă pentru această problemă, aşa cum este arătat în figura 34.4: calculul lui p şi t_s modulo o valoare potrivită q. Întrucât calculul lui p, t_0 şi recursivitatea (34.1) pot fi efectuate modulo q, observăm că p şi toate valorile t_s pot fi calculate modulo q în timpul $O(n+m)$. Pentru valoarea q este, de obicei, ales un număr prim, astfel încât $10q$ să poată fi reprezentat pe un cuvânt de memorie, ceea ce permite ca toate operaţiile necesare să fie efectuate cu precizie aritmetică simplă. În general, având dat un alfabet $\{0, 1, ..., d-1\}$, alegem q astfel încât dq să se poată reprezenta pe un cuvânt de memorie, şi modificăm ecuaţia recursivă (34.1), astfel încât să se calculeze modulo q, şi deci:

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q, \tag{34.2}$$

unde $h \equiv d^{m-1} \pmod q$ este valoarea pentru cifra "1" în poziţia cea mai semnificativă a ferestrei text de m-cifre.

Calitatea folosirii modulo q conferă rapiditate, deşi $t_s \equiv p \pmod q$ nu implică $t_s = p$. Pe de altă parte, dacă $t_s \neq p \pmod q$, atunci, bineînţeles, avem $t_s \neq p$, astfel încât deplasamentul s este incorect. Putem, astfel, folosi testul $t_s \equiv p \pmod q$ ca pe un test euristic rapid pentru determinarea deplasamentelor incorecte s. Orice deplasament s, pentru care $t_s \equiv p \pmod q$, trebuie să fie testat în plus pentru a vedea dacă s este, într-adevăr, corect sau avem doar o ***falsă potrivire***. Această testare poate fi făcută prin verificarea, explicită, a condiţiei $P[1..m] = T[s+1..s+m]$. Dacă q este suficient de mare, atunci, putem spera că potrivirile false apar destul de rar şi, astfel, costul verificărilor suplimentare este scăzut.

Aceste idei se vor clarifica urmărind procedura de mai jos. Datele de intrare pentru procedură sunt textul T, modelul P, baza d folosită (care de obicei este aleasă $|\Sigma|$) şi numărul prim q.

POTRIVIRE-RABIN-KARP(T, P, d, q)
1: $n \leftarrow lungime[T]$
2: $m \leftarrow lungime[P]$
3: $h \leftarrow d^{m-1} \bmod q$
4: $p \leftarrow 0$
5: $t_0 \leftarrow 0$
6: **pentru** $i \leftarrow 1$, m **execută**
7: $p \leftarrow (dp + P[i]) \bmod q$
8: $t_0 \leftarrow (dt_0 + T[i]) \bmod q$
9: **pentru** $s \leftarrow 0$, $n - m$ **execută**
10: **dacă** $p = t_s$ **atunci**
11: **dacă** $P[1..m]$=$T[s+1..s+m]$ **atunci**
12: tipăreşte "Modelul apare cu deplasamentul" s
13: **dacă** $s < n - m$ **atunci**
14: $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

Descriem, în continuare, modul în de funcţionare a procedurii POTRIVIRE-RABIN-KARP. Toate caracterele sunt interpretate ca cifre în baza d. Indicele lui t este folosit numai pentru claritate; programul funcţionează corect şi dacă toţi indicii sunt omişi. În linia 3, se iniţializează h cu valoarea de pe poziţia cea mai semnificativă dintr-o fereastră de m cifre. În liniile 4–8, se calculează p ca valoarea $P[1..m] \bmod q$ şi t_0 ca fiind valoarea lui $T[1..m] \bmod q$. Ciclul **pentru**, începând din linia 9, parcurge toate deplasamentele posibile s. Ciclul are următorul invariant: de fiecare dată când linia 10 este executată, $t_s = T[s+1..s+m] \bmod q$. Dacă, în linia 10, avem

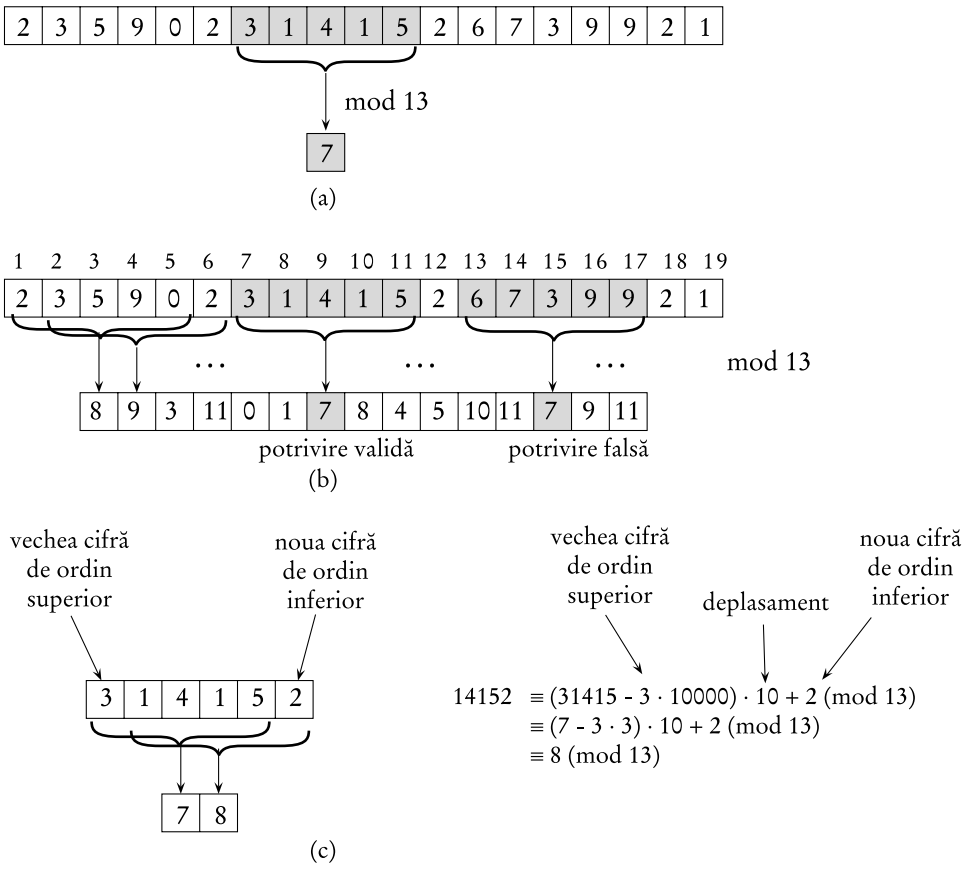


Figura 34.4 Algoritmul Rabin-Karp. Fiecare caracter este o cifră zecimală şi calculăm valorile modulo 13. **(a)** Un şir text. O fereastră de lungime 5 este reprezentată haşurat. Valoarea numerică a numărului haşurat este 7 calculată modulo 13. **(b)** Acelaşi şir text cu valori calculate modulo 13 pentru fiecare poziţie posibilă a ferestrei de lungime 5. Presupunând modelul $P = 31415$, căutăm ferestrele ale căror valori modulo 13 sunt 7, deoarece $31415 \equiv 7 \pmod{13}$. Sunt determinate şi haşurate două astfel de ferestre. Prima, începând cu poziţia 7 în text, este, într-adevăr, o apariţie a modelului, în timp ce a doua, începând cu poziţia 13 în text, este o falsă potrivire. **(c)** Calculând valoarea pentru o fereastră într-un timp constant, am dat valoarea pentru fereastra precedentă. Prima fereastră are valoarea 31415. Înlăturând cifra cea mai semnificativă, 3, deplasând spre stânga (înmulţind cu 10) şi, apoi, adunând, pe poziţia cea mai puţin semnificativă a cifra 2, obţinem noua valoare 14152. Toate calculele sunt efectuate modulo 13, astfel, valoarea pentru prima fereastră este 7, şi valoarea calculată pentru noua fereastră este 8.

$p = t_s$ (o "potrivire"), atunci, în linia 11 se verifică dacă $P[1..m] = T[s+1..s+m]$ pentru a elimina posibilitatea unei potriviri false. În linia 12 se tipăresc toate deplasamentele corecte determinate. Dacă $s < n - m$ (verificarea din linia 13), atunci ciclul **pentru** este executat cel puţin încă o dată, şi, astfel, linia 14 este executată pentru a asigura că invariantul ciclului este corect când se ajunge din nou la linia 10. Linia 14 calculează valoarea $t_{s+1} \bmod q$ din valoarea $t_s \bmod q$ într-un

timp constant, folosind direct ecuaţia (34.2).

Timpul de execuţie, în cel mai defavorabil caz, pentru algoritmul POTRIVIRE-RABIN-KARP este $\Theta((n-m+1)m)$ deoarece (ca în algoritmul naiv de potrivire a şirurilor) algoritmul Rabin-Karp verifică explicit fiecare deplasament corect. Dacă $P = \mathtt{a}^m$ şi $T = \mathtt{a}^n$, atunci verificările durează un timp $\Theta((n-m+1)m)$ deoarece fiecare dintre cele $n-m+1$ deplasamente posibile sunt corecte. (Subliniem, de asemenea, că timpul necesar pentru calculul lui $d^{m-1} \bmod q$, din linia a 3-a, şi ciclul din liniile 6–8 este $O(m) = O((n-m+1)m)$.) În multe aplicaţii, ne aşteptăm la puţine deplasamente corecte (probabil $O(1)$ dintre ele). Astfel, timpul de execuţie aşteptat pentru algoritm este $O(n+m)$, plus timpul necesar pentru procesarea potrivirilor false. Putem susţine o analiză euristică presupunând că reducerea valorilor modulo q acţionează ca o proiecţie aleatoare a lui Σ^* peste Z_q. (Vezi discuţia despre folosirea divizării pentru dispersie din secţiunea 12.3.1. Este dificil să formalizăm şi să demonstrăm o astfel de presupunere, deşi o modalitate viabilă este să presupunem că q este ales aleator, din mulţimea numerelor întregi, cu o valoare potrivită. Nu vom urmări aici această formalizare.) Ne putem aştepta ca numărul de potriviri false să fie $O(n/q)$, deoarece posibilitatea ca un t_s arbitrar să fie echivalent cu p, modulo q, poate fi estimată la $1/q$. Atunci, timpul de execuţie estimat pentru algoritmul Rabin-Karp este:

$$O(n) + O(m(v + n/q)),$$

unde v este numărul de deplasamente corecte. Acest timp de execuţie este $O(n)$ dacă alegem $q \geq m$. Aşadar, dacă numărul estimat de deplasamente corecte este mic ($O(1)$) şi numărul prim q este ales mai mare decât lungimea modelului, atunci, estimăm pentru procedura Rabin-Karp timpul de execuţie $O(n+m)$.

Exerciţii

34.2-1 Lucrând modulo $q = 11$, câte potriviri false se întâlnesc la execuţia algoritmului Rabin-Karp pentru textul $T = 3141592653589793$ şi pentru modelul $P = 26$?

34.2-2 Cum aţi extinde metoda Rabin-Karp la problema căutării într-un text a câte unei singure apariţii a fiecărui model dintr-o mulţime dată de k modele?

34.2-3 Arătaţi cum se extinde metoda Rabin-Karp pentru a rezolva problema căutării unui model dat $m \times m$ într-un şir de caractere $n \times n$. (Modelul poate fi deplasat vertical şi orizontal, dar nu poate fi rotit.)

34.2-4 Alice are o copie a unui fişier $A = \{a_{n-1}, a_{n-2}, ..., a_0\}$ de lungime n biţi, şi Bob are, de asemenea, un fişier $B = \{b_{n-1}, b_{n-2}, ..., b_0\}$ care conţine n biţi. Alice şi Bob doresc să ştie dacă fişierele lor sunt identice. Pentru a evita transmiterea întregului fişier A sau B, ei folosesc următoarea verificare probabilistică rapidă. Aleg împreună numărul $q > 1000n$ şi un întreg x aleator din mulţimea $\{0, 1, ..., q-1\}$. Apoi, Alice calculează:

$$A(x) = \left(\sum_{i=0}^{n} a_i x^i \right) \bmod q$$

şi în mod analog, Bob calculează $B(x)$. Demonstraţi că, dacă $A \neq B$, există, cel mult, o şansă la 1000 ca $A(x) = B(x)$, întrucât, dacă cele două fişiere sunt identice, $A(x)$ este neapărat acelaşi cu $B(x)$. (*Indicaţie:* vezi exerciţiul 33.4-4.)

34.3. Potrivirea şirurilor folosind automate finite

Mulţi algoritmi de potrivire a şirurilor construiesc un automat finit care caută în textul T toate apariţiile modelului P. Această secţiune prezintă o metodă pentru construirea unui astfel de automat. Aceste automate de potrivire a şirurilor sunt foarte eficiente: ele analizează fiecare caracter al textului *o singură dată*, consumând un timp constant pentru fiecare caracter. Timpul folosit – după ce se construieşte automatul – este, aşadar, $\Theta(n)$. Cu toate acestea, timpul pentru construirea automatului poate fi mare dacă Σ este mare. În secţiunea 34.4 se descrie o variantă inteligentă de abordare a acestei probleme.

Începem această secţiune cu definiţia unui automat finit. Examinăm apoi un automat special de potrivire a şirurilor şi arătăm cum poate fi el folosit pentru determinarea potrivirilor unui model într-un text. Discuţia include detalii privind simularea comportamentului unui automat de potrivire a şirurilor pe un text dat. În final vom arăta cum se construieşte automatul de potrivire a şirurilor pentru un model de intrare dat.

Automate finite

Un ***automat finit*** M este un cvintuplu $(Q, q_0, A, \Sigma, \delta)$, unde

- Q este o mulţime finită de ***stări***,
- $q_0 \in Q$ este ***starea de start***,
- $A \subseteq Q$ este o mulţime distinctă de ***stări de acceptare***,
- Σ este un ***alfabet de intrare*** finit,
- δ este o funcţie definită pe $Q \times \Sigma$ cu valori în Q, numită ***funcţie de tranziţie*** a automatului M.

Automatul finit începe în starea q_0 şi citeşte, unul câte unul, caracterele din şirul de intrare. Dacă automatul este în starea q şi citeşte caracterul de intrare a, trece ("execută o tranziţie") din starea q în starea $\delta(q, a)$. Atunci când starea curentă q a automatului aparţine mulţimii A, spunem că maşina M ***acceptă*** şirul citit până în acest moment. Dacă o intrare nu este acceptată, spunem că este ***respinsă***. Figura 34.5 ilustrează această difiniţie printr-un automat simplu cu două stări.

Automatul finit M induce o funcţie ϕ, numită ***funcţie de stare finală***, definită pe Σ^* cu valori în Q astfel încât $\phi(w)$ este starea în care rămâne M după parcurgerea şirului w. Astfel, M acceptă un şir w dacă şi numai dacă $\phi(w) \in A$. Funcţia ϕ este definită de relaţia recursivă

$$\begin{aligned} \phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \text{ pentru } w \in \Sigma^*, a \in \Sigma. \end{aligned}$$

Automate de potrivire a şirurilor

Există un automat de potrivire a şirurilor pentru fiecare model P; acest automat trebuie construit (pornind de la modelul dat) înainte de a putea fi folosit pentru căutare. Figura 34.6

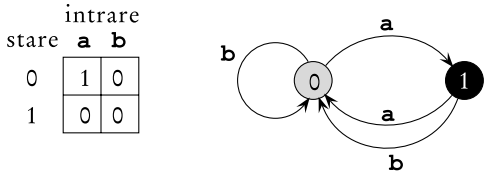


Figura 34.5 Un automat finit simplu, cu două stări, pentru care mulţimea stărilor este $Q = \{0, 1\}$, starea de start este $q_0 = 0$, iar alfabetul de intrare este $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. **(a)** O reprezentare vectorială a funcţiei de tranziţie δ. **(b)** O diagramă echivalentă a stărilor de tranziţie. Starea 1 este singura stare de acceptare (colorată cu negru). Arcele orientate reprezintă tranziţii. De exemplu, arcul de la starea 1 la starea 0, etichetat cu `b` indică $\delta(1, \mathtt{b}) = 0$. Acest automat acceptă acele şiruri care se termină cu un număr impar de `a`-uri. Mai exact, un şir x este acceptat dacă, şi numai dacă, $x = yz$, unde $y = \varepsilon$ sau y se termină cu un `b`, şi $z = \mathtt{a}^k$, unde k este impar. De exemplu, secvenţa de stări pe care acest automat le are pentru intrarea `abaaa` (inclusiv starea de start) este $\langle 0, 1, 0, 1, 0, 1\rangle$ şi, de aceea, acceptă această intrare. Pentru intrarea `abbaa`, secvenţa de stări este $\langle 0, 1, 0, 0, 1, 0\rangle$ şi deci, această intrare nu este acceptată.

ilustrează această construcţie pentru modelul $P = \mathtt{ababaca}$. În continuare, vom presupune că P este un model fixat; pentru uşurarea exprimării, în notaţii, nu vom indica toate dependenţele pentru P.

Pentru a specifica automatul de potrivire a şirurilor, corespunzător modelului dat $P[1..m]$, definim mai întâi funcţia σ, numită ***funcţie sufix***, corespunzătoare lui P. Funcţia σ este o proiecţie a lui Σ^* peste $\{0, 1, ..., m\}$, astfel încât $\sigma(x)$ este lungimea celui mai lung prefix al lui P, care este sufix pentru x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}.$$

Funcţia sufix σ este bine definită deoarece şirul vid $P_0 = \varepsilon$ este un sufix pentru orice şir. De exemplu, pentru modelul $P = \mathtt{ab}$, avem $\sigma(\varepsilon) = 0$, $\sigma(\mathtt{ccaca}) = 1$ si $\sigma(\mathtt{ccab}) = 2$. Pentru modelul P de lungime m, avem $\sigma(x) = m$ dacă şi numai dacă, $P \sqsupset x$. Din definiţia funcţiei sufix rezultă că, dacă $x \sqsupset y$, atunci $\sigma(x) \leq \sigma(y)$.

Definim automatul de potrivire a şirurilor pentru un model dat $P[1..m]$ după cum urmează.

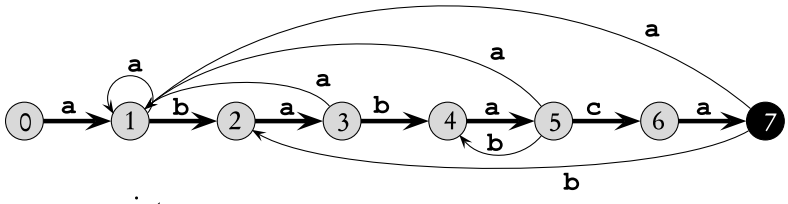
- Mulţimea stărilor Q este $\{0, 1, .., m\}$. Starea de start q_0 este starea 0, şi starea m este singura stare de acceptare.

- Funcţia de tranziţie δ este definită prin următoarea ecuaţie, pentru orice stare q şi orice caracter a:

$$\delta(q, a) = \sigma(P_q a). \tag{34.3}$$

Există o raţiune intuitivă pentru definirea $\delta(q, a) = \sigma(P_q a)$: maşina menţine un invariant pe parcursul operaţiilor ei:

$$\phi(T_i) = \sigma(T_i), \tag{34.4}$$



(a)

stare	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	–	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	–	a	b	a	b	a	b	a	c	a	b	a
$\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Figura 34.6 **(a)** O diagramă a stărilor de tranziţie pentru automatul de potrivire a şirurilor care acceptă toate şirurile terminate cu `ababaca`. Starea 0 este starea de start şi starea 7 (colorată cu negru) este singura stare de acceptare. Un arc orientat de la starea i la starea j, etichetat cu a, reprezintă $\delta(i, a) = j$. Arcele orientate spre dreapta, formând "coloana vertebrală" a automatului, reprezentate îngroşat, corespund potrivirilor corecte între model şi caracterele de intrare. Arcele orientate spre stânga corespund potrivirilor incorecte. Unele arce, corespunzând potrivirilor incorecte, nu sunt reprezentate; prin convenţie, dacă o stare i nu are arc de ieşire etichetat cu a pentru $a \in \Sigma$, atunci $\delta(i, a) = 0$. **(b)** Funcţia de tranziţie corespunzătoare δ şi şirul model $P =$ `ababaca`. Intrările corespunzând potrivirilor corecte între model şi caracterele de intrare sunt haşurate. **(c)** Funcţionarea automatului pentru textul $T =$ `abababacaba`. Sub fiecare caracter al textului $T[i]$, este dată starea $\phi(T_i)$ în care se află automatul după procesarea prefixului T_i. O apariţie a modelului este determinată, terminându-se în poziţia 9.

rezultat care este demonstrat mai jos în teorema 34.4. Cu alte cuvinte, aceasta înseamnă că după parcurgerea primelor i caractere din textul T, maşina este în starea $\phi(T_i) = q$, unde $q = \sigma(T_i)$ este lungimea celui mai lung sufix din T_i care este, de asemenea, un prefix al modelului P. Dacă următorul caracter prelucrat este $T[i+1] = a$, atunci maşina trebuie să facă o tranziţie la starea $\sigma(T_{i+1}) = \sigma(T_i a)$. Demonstraţia teoremei arată că $\sigma(T_i a) = \sigma(P_q a)$. Aşadar, pentru a calcula lungimea celui mai lung sufix din $T_i a$ care este un prefix al lui P, putem calcula cel mai lung sufix al lui $P_q a$ care este un prefix al lui P. În fiecare stare, maşina trebuie să cunoască doar lungimea celui mai lung prefix al lui P care este un sufix a ceea ce a fost citit până aici. De aceea, atribuirea $\delta(q, a) = \sigma(P_q a)$ păstrează invariantul dorit (34.4). Această argumentare va fi susţinută riguros mai târziu.

În automatul de potrivire a şirurilor din figura 34.6, de exemplu, avem $\delta(5, \mathtt{b}) = 4$. Aceasta rezultă din faptul că, dacă automatul citeşte un `b` în starea $q = 5$, atunci P_q`b` = `ababab` şi cel mai lung prefix al lui P care este, totodată, sufix al lui `ababab` este $P_4 =$ `abab`.

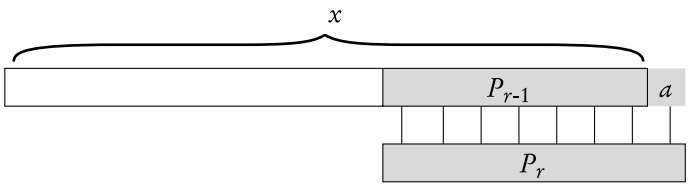


Figura 34.7 O ilustrare pentru demonstraţia lemei 34.2. Figura arată că $r \leq \sigma(x)+1$, unde $r = \sigma(xa)$.

Pentru a clarifica modul de funcţionare a automatului de potrivire a şirurilor, dăm, în continuare, un program simplu şi eficient pentru simularea comportamentului acestuia (reprezentat prin funcţia sa de tranziţie δ) în căutarea aparıţiilor unui model P de lungime m într-un text de intrare $T[1..n]$. La fel ca la toate automatele de potrivire a şirurilor pentru un model de lungime m, mulţimea stărilor Q este $\{0, 1, ..., m\}$ şi starea de start este 0; singura stare de acceptare este starea m.

AUTOMAT-FINIT-DE-POTRIVIRE(T,δ,m)
1: $n \leftarrow lungime[T]$
2: $q \leftarrow 0$
3: **pentru** $i \leftarrow 1$, n **execută**
4: $q \leftarrow \delta(q, T[i])$
5: **dacă** $q = m$ **atunci**
6: $s \leftarrow i - m$
7: tipăreşte "Modelul apare cu deplasamentul" s

Datorită structurii de ciclu simplu al algoritmului AUTOMAT-FINIT-DE-POTRIVIRE, timpul de execuţie, pentru un text de lungime n, este $O(n)$. Acest timp de execuţie nu include timpul necesar efectuării calculului funcţiei de tranziţie δ. Această problemă va fi abordată mai târziu, după ce se demonstrează că procedura AUTOMAT-FINIT-DE-POTRIVIRE funcţionează corect.

Analizăm funcţionarea automatului pe un text de intrare $T[1..n]$. Vom demonstra că automatul ajunge în starea $\sigma(T_i)$ după prelucrarea caracterului $T[i]$. Întrucât $\sigma(T_i) = m$ dacă, şi numai dacă, $P \sqsupset T_i$, maşina ajunge în starea de aşteptare m dacă şi numai dacă modelul P tocmai a fost prelucrat. Pentru a demonstra aceasta vom folosi următoarele două leme referitoare la funcţia sufix σ.

Lema 34.2 (Inegalitatea funcţiei sufix) Pentru orice şir x şi orice caracter a, avem $\sigma(xa) \leq \sigma(x) + 1$.

Demonstraţie. În figura 34.7, considerăm $r = \sigma(xa)$. Dacă $r = 0$, atunci concluzia $r \leq \sigma(x) + 1$ este imediată, deoarece funcţia $\sigma(x)$ este pozitivă. În continuare presupunem că $r > 0$. Deci, $P_r \sqsupset xa$ conform definiţiei lui σ. Astfel, $P_{r-1} \sqsupset x$, la eliminarea lui a de la sfârşitul lui P_r şi de la sfârşitul lui xa. Rezultă că $r - 1 \leq \sigma(x)$ întrucât $\sigma(x)$ este cel mai mare k pentru care $P_k \sqsupset x$. ∎

Lema 34.3 (Lema de recursivitate a funcţiei sufix) Pentru orice şir x şi orice caracter a, dacă avem $q = \sigma(x)$, atunci $\sigma(xa) = \sigma(P_q a)$.

Demonstraţie. Din definiţia lui σ avem a $P_q \sqsupset x$. În figura 34.8 se ilustrează că $P_q a \sqsupset xa$. Dacă alegem $r = \sigma(xa)$, atunci, conform lemei 34.2, $r \leq q + 1$. Întrucât $P_q a \sqsupset xa$, $P_r \sqsupset xa$ şi $|P_r| \leq$

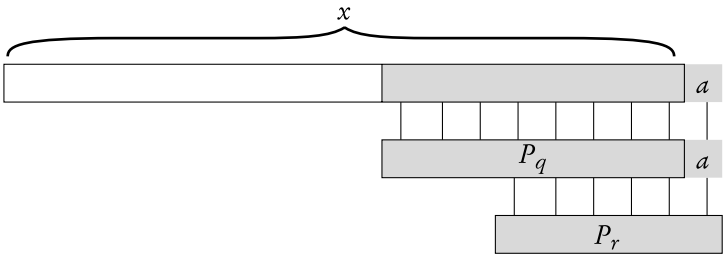


Figura 34.8 O ilustrare pentru demonstraţia lemei 34.3. Figura arată că $r = \sigma(P_q a)$, unde $q = \sigma(x)$ şi $r = \sigma(xa)$.

$|P_q a|$, conform lemei 34.1, rezultă că $P_r \sqsupset P_q a$. Deci, $r \leq \sigma(P_q a)$ înseamnă că $\sigma(xa) \leq \sigma(P_q a)$. Dar, de asemenea, avem că $\sigma(P_q a) \leq \sigma(xa)$, pentru că $P_q a \sqsupset xa$. Rezultă că $\sigma(xa) = \sigma(P_q a)$. ■

Avem toate datele necesare pentru a demonstra teorema principală ce caracterizează comportamentul unui automat de potrivire a şirurilor pentru un text de intrare dat. Aşa cum am observat mai sus, această teoremă arată că, la fiecare pas, este suficient ca automatul să cunoască lungimea celui mai lung prefix al modelului care este un sufix a ceea ce a fost prelucrat până atunci.

Teorema 34.4 Dacă ϕ este funcţia de stare finală a automatului de potrivire a şirurilor pentru un model dat P şi un text de intrare $T[1..n]$, atunci

$$\phi(T_i) = \sigma(T_i)$$

pentru $i = 0, 1, ..., n$.

Demonstraţie. Demonstraţia se face prin inducţie după i. Pentru $i = 0$, teorema este imediată, deoarece $T_0 = \varepsilon$. Deci, $\phi(T_0) = \sigma(T_0) = 0$.

Acum, presupunem că $\phi(T_i) = \sigma(T_i)$ şi demonstrăm că $\phi(T_i + 1) = \sigma(T_{i+1})$. Notăm cu q pe $\phi(T_i)$ şi cu a pe $T[i + 1]$. Atunci,

$$\begin{array}{rcll}
\phi(T_{i+1}) & = & \phi(T_i, a) & \text{(din definiţia lui } T_{i+1} \text{ şi } a) \\
& = & \delta(\phi(T_i), a) & \text{(din definiţia lui } \phi) \\
& = & \delta(q, a) & \text{(din definiţia lui } q) \\
& = & \sigma(P_q a) & \text{(din definiţia (34.3) a lui } \delta) \\
& = & \sigma(T_i a) & \text{(din lema 34.3 şi din inducţie)} \\
& = & \sigma(T_{i+1}) & \text{(din definiţia lui } T_{i+1})
\end{array}$$

Teorema este demonstrată prin inducţie. ■

Conform teoremei 34.4, dacă maşina intră în starea q în linia 4, atunci q este cea mai mare valoare, astfel încât $P_q \sqsupset T_i$. Avem $q = m$ în linia 5 dacă, şi numai dacă, o apariţie a modelului P, tocmai, a fost prelucrată. În concluzie, AUTOMAT-FINIT-DE-POTRIVIRE funcţionează corect.

Calculul funcţiei de tranziţie

Procedura următoare calculează funcţia de tranziţie δ pornind de la modelul dat $P[1..m]$.

CALCUL-FUNCŢIE-DE-TRANZIŢIE(P, Σ)
1: $m \leftarrow lungime[P]$
2: **pentru** $q \leftarrow 0$, m **execută**
3: **pentru** fiecare caracter $a \in \Sigma$ **execută**
4: $k \leftarrow \min(m+1, q+2)$
5: **repetă**
6: $k \leftarrow k-1$
7: **până când** $P_k \sqsupset P_q a$
8: $\delta(q,a) \leftarrow k$
9: **returnează** δ

Această procedură calculează $\delta(q,a)$ într-o manieră simplă, conform definiţiei sale. Ciclurile imbricate, care încep din linile 2 şi 3, parcurg toate stările q şi caracterele a, iar în liniile 4–7 se atribuie pentru $\delta(q,a)$ cea mai mare valoare a lui k, astfel încât $P_k \sqsupset P_q a$. Codul funcţiei începe cu cea mai mare valoare posibilă pentru k, care este $\min(m, q+1)$ şi decrementează k până când $P_k \sqsupset P_q a$.

Timpul de execuţie pentru CALCUL-FUNCŢIE-DE-TRANZIŢIE este $O(m^3|\Sigma|)$ deoarece ciclul exterior contribuie cu un factor $m|\Sigma|$, ciclul, **repetă**, din interior se poate executa de cel mult $m+1$ ori şi testul $P_k \sqsupset P_q a$, din linia 6, poate necesita compararea a până la m caractere. Există proceduri mult mai rapide; timpul necesar pentru calcularea lui δ, pornind de la P, poate fi îmbunătăţit la $O(m|\Sigma|)$ prin utilizarea unor informaţii deduse inteligent despre modelul P (vezi exerciţiu 34.4-6). Cu această procedură îmbunătăţită pentru calcularea lui δ, timpul total de execuţie pentru determinarea tuturor aparıţiilor unui model de lungime m, într-un text de lungime n, peste un alfabet Σ este $O(n + m|\Sigma|)$.

Exerciţii

34.3-1 Construiţi automatul de potrivire a şirurior pentru modelul $P =$ `aabab` şi ilustraţi funcţionarea sa pe textul $T =$ `aaababaabaababaab`.

34.3-2 Desenaţi o diagramă stare-tranziţie pentru un automat de potrivire a şirurilor pentru modelul `ababbabbababbababbabb` peste alfabetul $\Sigma = \{$`a`,`b`$\}$.

34.3-3 Spunem că un model P ***nu se poate suprapune*** dacă $P_k \sqsupset P_q$ implică $k = 0$ sau $k = q$. Descrieţi diagrama stare-tranziţie a automatului de potrivire a şirurilor pentru un model care nu se poate suprapune.

34.3-4 ⋆ Fiind date modelele P şi P', descrieţi modul în care se construieşte un automat finit care determină toate aparıţiile ambelor modele. Încercaţi să minimizaţi numărul de stări ale automatului.

34.3-5 Fiind dat modelul P conţinând caractere de discontinuitate (vezi exerciţiul 34.1-5), arătaţi cum se construieşte un automat finit care poate determina o aparıţie a lui P într-un text T într-un timp $O(n)$, unde $n = |T|$.

34.4. Algoritmul Knuth-Morris-Pratt

Prezentăm acum un algoritm de potrivire a şirurilor cu timpul de execuţie liniar datorat lui Knuth, Morris şi Pratt. Algoritmul lor realizează un timp de execuţie $\Theta(n+m)$ prin evitarea calculului întregii funcţii de tranziţie δ şi potrivirea modelului folosind chiar o funcţie auxiliară $\pi[1..m]$, precalculată pornind de la model în timpul $O(m)$. Şirul π permite funcţiei de tranziţie δ să fie calculată eficient (în sens de amortizare) "din zbor", aşa cum este necesar. În general, pentru orice stare $q = 0, 1, ..., m$ şi orice caracter $a \in \Sigma$, valoarea $\pi[q]$ conţine informaţia care este independentă de a şi este necesară pentru calcularea lui $\delta(q, a)$. (Această remarcă va fi clarificată mai târziu.) Deoarece şirul π are doar m elemente şi δ are $O(m|\Sigma|)$ elemente, timpul de preprocesare îl reducem cu un factor Σ calculându-l pe π, în loc de δ.

Funcţia prefix pentru un model

Funcţia prefix pentru un model încapsulează informaţii despre potrivirea modelului cu deplasamentele lui. Aceste informaţii pot fi folosite pentru a evita testarea deplasamentelor nefolositoare în algoritmul naiv de potrivire a şirurilor sau pentru a evita precalcularea lui δ pentru un automat de potrivire a şirurilor.

Analizăm algoritmul naiv de potrivire a şirurilor. În figura 34.9(a) se arată un deplasament particular s al unui şablon conţinând modelul $P =$ `ababaca` pentru textul T. Pentru acest exemplu, $q = 5$ dintre caractere s-au potrivit cu succes, dar al 6-lea caracter nu s-a potrivit cu caracterul corespunzător din text. Informaţia că primele q caractere au fost potrivite cu succes determină caracterele text corespunzătoare. Cunoscând aceste q caractere din text, putem să determinăm imediat că anumite deplasamente sunt incorecte. În exemplul din figură, deplasamentul $s+1$ este sigur incorect deoarece primul caracter al modelului, un `a`, va fi aliniat cu caracterul din text care se ştie că se potriveşte cu al doilea caracter al modelului, un `b`. Deplasamentul $s+2$, prezentat în cadrul (b) al figurii, aliniază primele trei caractere ale modelului cu trei caractere din text care sigur se potrivesc. În general, este necesar să ştim răspunsul la următoarea întrebare:

Ştiind că modelul $P[1..q]$ se potriveşte cu caracterele text $T[s+1..s+q]$, care este cel mai mic deplasament $s' > s$, astfel încât

$$P[1..k] = T[s'+1..s'+k], \tag{34.5}$$

unde $s' + k = s + q$?

Un astfel de deplasament s' este primul deplasament mai mare decât s care nu este neapărat incorect conform informaţiilor despre $T[s+1..s+q]$. În cel mai bun caz avem că $s' = s + q$, şi deplasamentele $s+1, s+2, ..., s+q-1$ sunt toate eliminate. În orice caz, la noul deplasament s' nu mai trebuie să comparăm primele k caractere ale lui P cu caracterele corespunzătoare ale lui T deoarece ecuaţia (34.5) ne asigură că ele se potrivesc.

Informaţia necesară poate fi precalculată comparând modelul cu el însuşi, aşa cum este ilustrat în figura 34.9(c). Deoarece $T[s'+1..s'+k]$ face parte din porţiunea de text cunoscută, el este un sufix al şirului P_q. Deci, ecuaţia (34.5) poate fi interpretată ca o cerere pentru cel mai mare $k < q$, astfel încât $P_k \sqsupset P_q$. Atunci, următorul potenţial deplasament corect este $s' = s + (q - k)$. Se întâmplă să fie mai convenabil să memorăm un număr de k caractere care

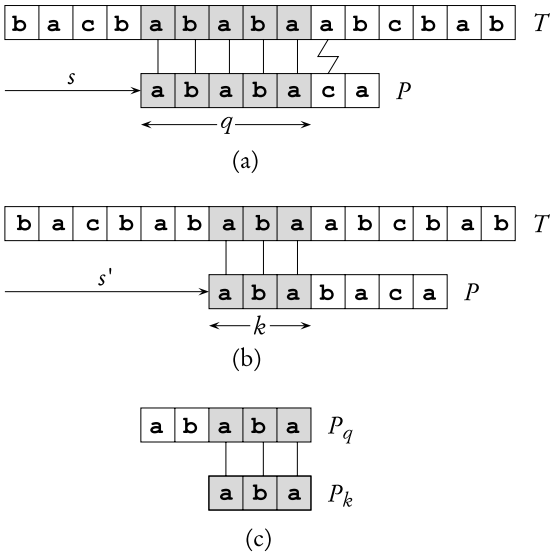


Figura 34.9 Funcţia prefix π. **(a)** Modelul $P =$ `ababaca` este aliniat cu textul T, astfel încât primele $q = 5$ caractere se potrivesc. Caracterele care se potrivesc sunt unite prin linii verticale şi sunt haşurate. **(b)** Folosind numai informaţiile noastre despre cele 5 caractere care se potrivesc, putem deduce că un deplasament $s+1$ este incorect, dar că un deplasament $s' = s+2$ se potriveşte cu tot ce ştim despre text şi deci există posibilitatea să fie corect. **(c)** Informaţia utilă pentru astfel de deducţii poate fi precalculată comparând modelul cu el însuşi. Aici, se observă că cel mai lung prefix al lui P, care este şi un sufix al lui P_5, este P_3. Această informaţie este precalculată şi reprezentată în şirul π, astfel $\pi[5] = 3$. Dacă q caractere s-au potrivit cu succes la deplasamentul s, următorul deplasament corect posibil este la $s' = s + (q - \pi[q])$.

se potrivesc la noul deplasament s', decât să memorăm, de exemplu, $s' - s$ caractere. Această informaţie poate fi utilizată pentru a mări viteza atât la algoritmul naiv de potrivire a şirurilor, cât şi la procedura de potrivire folosind automatul finit.

Formalizăm, în continuare, precalculul necesar. Fiind dat un model $P[1..m]$, ***funcţia prefix*** pentru modelul P este $\pi : \{1, 2, ..., m\} \to \{0, 1, ..., m-1\}$, astfel încât

$$\pi[q] = \max\{k : k < q \text{ şi } P_k \sqsupset P_q\}.$$

Astfel, $\pi[q]$ este lungimea celui mai lung prefix al lui P care este un sufix potrivit pentru P_q. Ca un alt exemplu, în figura 34.10(a) este prezentată întreaga funcţie prefix π pentru modelul `abababaca`.

Algoritmul de potrivire Knuth-Morris-Pratt, este prezentat în pseudocod, prin procedura POTRIVIRE-KMP. Veţi vedea că este asemănător procedurii aproape ca şi AUTOMAT-FINIT-DE-POTRIVIRE. Pentru calculul lui π, procedura POTRIVIRE-KMP apelează procedura auxiliară CALCUL-FUNCŢIE-PREFIX.

```
POTRIVIRE-KMP(T,P)
 1: n ← lungime[T]
 2: m ← lungime[P]
 3: π ← CALCUL-FUNCŢIE-PREFIX(P)
 4: q ← 0
 5: pentru i ← 1, n execută
 6:    cât timp q > 0 şi P[q + 1] ≠ T[i] execută
 7:       q ← π[q]
 8:    dacă P[q + 1] = T[i] atunci
 9:       q ← q + 1
10:    dacă q = m atunci
11:       tipăreşte "Modelul apare cu deplasamentul" i − m
12:       q ← π[q]
```

```
CALCUL-FUNCŢIE-PREFIX(P)
 1: m ← lungime[P]
 2: π[1] ← 0
 3: k ← 0
 4: pentru q ← 2, m execută
 5:    cât timp k > 0 şi P[k + 1] ≠ P[q] execută
 6:       k ← π[k]
 7:    dacă P[k + 1] = P[q] atunci
 8:       k ← k + 1
 9:    π[q] ← k
10: returnează π
```

Începem cu analiza timpului de execuţie pentru aceste proceduri. Demonstrarea faptului că aceste proceduri sunt corecte va fi ceva mai complicată.

Analiza timpului de execuţie

Folosind o analiză amortizată (vezi capitolul 18), timpul de execuţie pentru funcţia CALCUL-FUNCŢIE-PREFIX este $O(m)$. Asociem un potenţial k cu starea k curentă a algoritmului. Acest potenţial are o valoare iniţială 0, în linia 3. Linia 6 decrementează k, ori de câte ori este executată, deoarece $\pi[k] < k$. Deoarece $\pi[k] \geq 0$ pentru orice k, k nu poate deveni negativ. Singura linie care mai afectează valoarea lui k este linia 8, unde se incrementează k, cel mult o dată, în timpul fiecărei iteraţii a ciclului **pentru**. Deoarece $k < q$ după intrarea în ciclul **pentru** şi deoarece q este incrementat la fiecare iteraţie a ciclului **pentru**, întotdeauna, va fi adevărată relaţia $k < q$. (Aceasta şi linia 9 justifică cerinţa $\pi[q] < q$.) Pentru fiecare iteraţie a ciclului **cât timp** din linia 6, putem decrementa funcţia potenţial, întrucât $\pi[k] < k$. Linia 8 incrementează funcţia potenţial cel mult o dată, deci costul amortizat al corpului ciclului din liniile 5–9 este $O(1)$. Deoarece numărul de iteraţii ale ciclului exterior este $O(m)$ şi deoarece funcţia potenţial finală este cel puţin la fel de mare ca funcţia potenţial iniţială, timpul total de execuţie, în cel mai rău caz, pentru CALCUL-FUNCŢIE-PREFIX este $O(m)$.

Algoritmul Knuth-Morris-Pratt se execută în timp $O(m+n)$. Apelul funcţiei CALCUL-FUNCŢIE-PREFIX consumă un timp $O(m)$, aşa cum tocmai am arătat, şi o analiză amortizată similară,

folosind ca funcţie potenţial valoarea q, arată că restul algoritmului POTRIVIRE-KMP necesită timpul $O(n)$.

Comparativ cu AUTOMAT-FINIT-DE-POTRIVIRE, prin folosirea lui π în locul lui δ, am redus timpul pentru preprocesarea modelului de la $O(m|\Sigma|)$ la $O(m)$, păstrând, simultan, timpul actual de potrivire limitat la $O(m+n)$.

Corectitudinea calculului funcţiei prefix

Începem cu o lemă esenţială care ne arată că, prin iterarea funcţiei prefix π, putem enumera toate prefixele P_k care sunt sufixe ale prefixului P_q dat. Fie

$$\pi^*[q] = \{q, \pi[q], \pi^2[q], \pi^3[q], ..., \pi^t[q]\},$$

unde $\pi^i[q]$ este definit ca funcţie compusă, astfel încât $\pi^0[q] = q$ şi $\pi^{i+1}[q] = \pi[\pi^i[q]]$ pentru $i > 1$ şi unde secvenţa $\pi^*[q]$ se termină când se obţine $\pi^t[q] = 0$.

Lema 34.5 (Lema de iteraţie a funcţiei prefix) Fie P un model de lungime m cu funcţia prefix π. Atunci, pentru $q = 1, 2, .., m$, avem $\pi^*[q] = \{k : P_k \sqsupset P_q\}$.

Demonstraţie. Arătăm, mai întâi, că:

$$i \in \pi^*[q] \text{ implică } P_i \sqsupset P_q \tag{34.6}$$

Dacă $i \in \pi^*[q]$, atunci, pentru un u oarecare, $i = \pi^u[q]$. Demonstrăm ecuaţia (34.6) prin inducţie după u. Pentru $u = 0$, avem $i = q$, şi cerinţa este îndeplinită deoarece $P_q \sqsupset P_q$. Folosind relaţia $P_{\pi[i]} \sqsupset P_i$ şi tranzitivitatea pentru relaţia $\sqsupset$, este fixată cerinţa pentru orice i din $\pi^*[q]$. Deci, $\pi^*[q] \subseteq \{k : P_k \sqsupset P_q\}$.

Demonstrăm că $\{k : P_k \sqsupset P_q\} \subseteq \pi^*[q]$ prin metoda reducerii la absurd. Presupunem că există un întreg în mulţimea $\{k : P_k \sqsupset P_q\} - \pi^*[q]$ şi fie j cel mai mare astfel de întreg. Deoarece q aparţine mulţimii $\{k : P_k \sqsupset P_q\} \cap \pi^*[q]$, avem $j < q$, aşadar fie j' cel mai mic întreg din $\pi^*[q]$ care este mai mare decât j. (Putem alege $j' = q$ dacă nu există alt număr în $\pi^*[q]$ mai mare decât j.) Avem $P_j \sqsupset P_q$ deoarece $j \in \{k : P_k \sqsupset P_q\}$, $P_{j'} \sqsupset P_q$ deoarece $j' \in \pi^*[q]$; astfel, $P_j \sqsupset P_{j'}$ conform lemei 34.1. Mai mult, j este cea mai mare valoare cu această proprietate. De aceea, trebuie să avem $\pi[j'] = j$ şi, astfel, $j \in \pi^*[q]$. Această contradicţie demonstrează lema. ■

Figura 34.10 ilustrează această lemă.

Algoritmul CALCUL-FUNCŢIE-PREFIX calculează $\pi[q]$ pentru fiecare $q = 1, 2, ..., m$. Calculul lui $\pi[1] = 0$ din linia 2 a funcţiei CALCUL-FUNCŢIE-PREFIX este corect deoarece $\pi[q] < q$ pentru orice q. Pentru a demonstra că funcţia CALCUL-FUNCŢIE-PREFIX calculează corect valoarea lui $\pi[q]$ pentru orice $q > 1$, vom folosi următoarea lemă şi corolarele acesteia.

Lema 34.6 Fie P un model de lungime m şi fie π funcţia prefix pentru P. Dacă $\pi[q] > 0$ pentru $q = 1, 2, ..., m$, atunci $\pi[q] - 1 \in \pi^*[q-1]$.

Demonstraţie. Dacă $k = \pi[q] > 0$, atunci $P_k \sqsupset P_q$, astfel $P_{k-1} \sqsupset P_{q-1}$ (eliminând ultimul caracter din P_k şi P_q). Deci, $k - 1 \in \pi^*[q-1]$ din lema 34.5. ■

Pentru $q = 2, 3, ..., m$, definim submulţimea $E_{q-1} \subseteq \pi^*[q-1]$ prin

$$E_{q-1} = \{k : k \in \pi^*[q-1] \text{ şi } P[k+1] = P[q]\}.$$

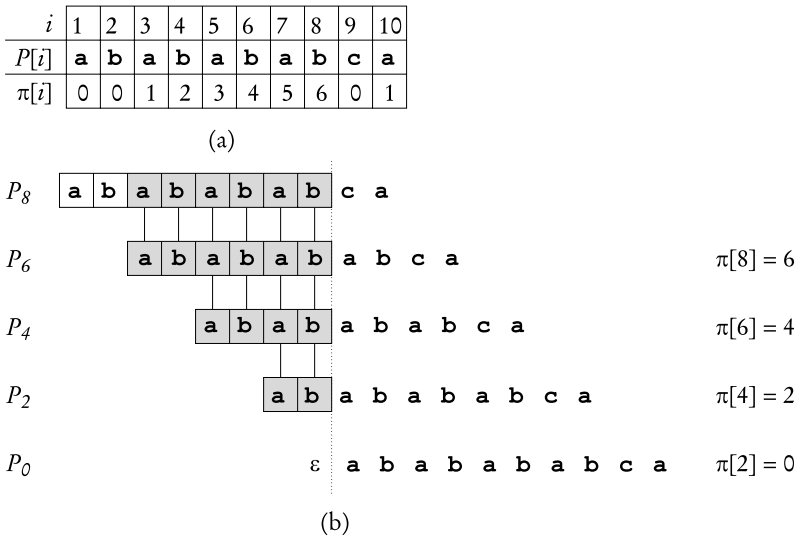


Figura 34.10 O ilustrare a lemei 34.5 pentru modelul $P =$ `abababa bca` şi $q = 8$. **(a)** Funcţia π pentru modelul dat. Întrucât $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$ şi $\pi[2] = 0$, iterând π, obţinem $\pi^*[8] = \{8, 6, 4, 2, 0\}$. **(b)** Deplasăm şablonul care conţine modelul P spre dreapta şi reţinem când un prefix P_k din P se potriveşte cu un sufix al lui P_8; aceasta se întâmplă pentru $k = 6, 4, 2$ şi 0. În figură, prima linie prezintă modelul P. Verticala punctată este trasată imediat după P_8. Liniile succesive arată toate deplasările lui P pentru care un prefix P_k al lui P se potriveşte cu un sufix al lui P_8. Caracterele care se potrivesc sunt haşurate. Liniile verticale unesc caracterele care se potrivesc. Astfel, $\{k : P_k \sqsupset P_q\} = \{8, 6, 4, 2, 0\}$. Lema afirmă că pentru orice q avem $\pi^*[q] = \{k : P_k \sqsupset P_q\}$.

Mulţimea E_{q-1} conţine acele valori k pentru care $P_k \sqsupset P_{q-1}$ (din lema 34.5); deoarece $P[k+1] = P[q]$, există, de asemenea, cazul în care pentru aceste valori ale lui k, $P_{k+1} \sqsupset P_q$. Intuitiv, E_{q-1} conţine acele valori $k \in \pi^*[q-1]$ care ne permite să extindem P_k la P_{k+1} şi să obţinem un sufix al lui P_q. ∎

Corolarul 34.7 Fie P un model de lungime m şi fie π funcţia prefix pentru modelul P. Pentru $q = 2, 3, .., m$

$$\pi[q] = \begin{cases} 0 & \text{dacă } E_{q-1} = \emptyset, \\ 1 + \max\{k \in E_{q-1}\} & \text{dacă } E_{q-1} \neq \emptyset. \end{cases}$$

Demonstraţie. Dacă $r = \pi[q]$, atunci $P_r \sqsupset P_q$ şi, deci, $r \geq 1$ implică $P[r] = P[q]$. Aplicând lema 34.6, dacă $r \geq 1$, atunci

$$r = 1 + \max\{k \in \pi^*[q-1] : P[k+1] = P[q]\}.$$

Dar mulţimea maximă este chiar E_{q-1}, deci $r = 1 + \max\{k \in E_{q-1}\}$, şi E_{q-1} este nevidă. Dacă $r = 0$, nu există $k \in \pi^*[q-1]$ pentru care să putem extinde P_k la P_{k+1} şi să putem obţine un sufix al lui P_q, deorece atunci am avea $\pi[q] > 0$, Deci , $E_{q-1} = \emptyset$. ∎

Astfel, am demonstrat faptul că funcţia CALCUL-FUNCŢIE-PREFIX calculează corect funcţia π. În procedura CALCUL-FUNCŢIE-PREFIX, la începutul fiecărei iteraţii a ciclului **pentru** din

liniile 4–9, avem $k = \pi[q-1]$. Această condiţie este impusă prin liniile 2 şi 3 la prima intrare în ciclu şi rămâne adevărată la fiecare iteraţie datorită liniei 9. Liniile 5–8 modifică valoarea lui k, astfel încât să devină valoarea corectă a lui $\pi[q]$. Ciclul din liniile 5–6 caută printre toate valorile $k \in \pi^*[q-1]$ până când este găsită una pentru care $P[k+1] = P[q]$; în acest moment k este cea mai mare valoare din mulţimea E_{q-1}, astfel, conform corolarului 34.7, putem atribui lui $\pi[q]$ valoarea $k+1$. Dacă nu este găsit un astfel de k, $k = 0$ în liniile 7–9, şi atribuim lui $\pi[q]$ valoarea 0. Cu aceasta am demonstrat corectitudinea algoritmului CALCUL-FUNCŢIE-PREFIX.

Corectitudinea algoritmului KMP

Procedura POTRIVIRE-KMP poate fi privită ca o reimplementare a procedurii AUTOMAT-FINIT-DE-POTRIVIRE. Mai precis, vom arăta că liniile 6–9 din POTRIVIRE-KMP sunt echivalente cu linia 4 din AUTOMAT-FINIT-DE-POTRIVIRE, care atribuie $\delta(q, T[i])$ lui q. În loc să memorăm valoarea $\delta(q, T[i])$, aceasta este recalculată, la nevoie, din π. O dată argumentat faptul că procedura POTRIVIRE-KMP simulează comportamentul procedurii AUTOMAT-FINIT-DE-POTRIVIRE, corectitudinea procedurii POTRIVIRE-KMP reiese implicit (totuşi vom vedea imediat de ce este necesară linia 12 în POTRIVIRE-KMP).

Corectitudinea procedurii POTRIVIRE-KMP rezultă din afirmaţia că $\delta(q, T[i]) = 0$ sau $\delta(q, T[i]) - 1 \in \pi^*[q]$. Pentru a verifica această afirmaţie, fie $k = \delta(q, T[i])$. Atunci, din definiţia lui δ şi σ, avem $P_k \sqsupset P_q T[i]$. De aceea, $k = 0$ sau $k \geq 1$, şi $P_{k-1} \sqsupset P_q$ prin eliminarea ultimelor caractere din P_k şi $P_q T[i]$ (în acest caz $k - 1 \in \pi^*[q]$). Aşadar, sau $k = 0$ sau $k - 1 \in \pi^*[q]$, rezultă că afirmaţia este adevărată.

Afirmaţia este folosită după cum urmează. Notăm cu q' valoarea lui q la intrarea în linia 6. Folosim echivalenţa $\pi^*[q] = \{k : P_k \sqsupset P_q\}$ pentru a justifica iteraţia $q \leftarrow \pi[q]$ care enumeră elementele $\{k : P_k \sqsupset P_{q'}\}$. Liniile 6–9 determină $\delta(q', T[i])$ prin examinarea elementelor lui $\pi^*[q']$ în ordine descrescătoare. Codul utilizează afirmaţia pentru a începe cu $q = \phi(T_{i-1}) = \sigma(T_{i-1})$ şi execută iteraţia $q \leftarrow \pi[q]$ până când este găsit un q, astfel încât $q = 0$ sau $P[q+1] = T[i]$. În primul caz, $\delta(q', T[i]) = 0$; în al doilea caz, q este elementul maxim în $E_{q'}$, aşa că $\delta(q', T[i]) = q+1$ conform corolarului 34.7.

Linia 12 din POTRIVIRE-KMP este necesară pentru a evita o posibilă referire la $P[m+1]$, în linia 6, după ce a fost determinată o potrivire a lui P. (Argumentul $q = \sigma(T_{i-1})$ rămâne corect. La următoarea execuţie a liniei 6, conform indicaţiei date în exerciţiul 34.4-6: $\delta(m, a) = \delta(\pi[m], a)$ sau, în mod echivalent, $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ oricare ar fi $a \in \Sigma$.) Ultimul argument pentru corectitudinea algoritmului Knuth-Morris-Pratt rezultă din corectitudinea procedurii AUTOMAT-FINIT-DE-POTRIVIRE, deoarece acum observăm că procedura POTRIVIRE-KMP simulează comportamentul procedurii AUTOMAT-FINIT-DE-POTRIVIRE.

Exerciţii

34.4-1 Calculaţi funcţia prefix π pentru modelul `ababbabbababbababbabb` când alfabetul este $\Sigma = \{\texttt{a}, \texttt{b}\}$.

34.4-2 Daţi o limită superioară pentru dimensiunea lui $\pi^*[q]$ ca funcţie de q. Daţi un exemplu pentru a arăta că limita dată este corectă.

34.4-3 Explicaţi cum se determină apariţia modelului P în textul T, examinând funcţia π pentru şirul PT (concatenarea lui P şi T de lungime $m + n$).

34.4-4 Arătaţi cum se îmbunătăţeşte procedura POTRIVIRE-KMP înlocuind π din linia 7 (dar nu şi din linia 12) cu π', unde π' este definit recursiv pentru $q = 1, 2, ..., m$ prin ecuaţia

$$\pi'[q] = \begin{cases} 0 & \text{dacă } \pi[q] = 0, \\ \pi'[\pi[q]] & \text{dacă } \pi[q] \neq 0 \text{ şi } P[\pi[q]+1] = P[q+1], \\ \pi[q] & \text{dacă } \pi[q] \neq 0 \text{ şi } P[\pi[q]+1] \neq P[q+1]. \end{cases}$$

Explicaţi de ce algoritmul modificat este corect şi în ce sens această modificare constituie o îmbunătăţire.

34.4-5 Daţi un algoritm de timp liniar pentru a determina dacă un text T este o rotaţie ciclică a altui şir T'. De exemplu, *arc* şi *car* sunt fiecare o rotaţie ciclică a celuilalt.

34.4-6 $\star$ Daţi un algoritm eficient pentru calculul funcţiei de tranziţie δ pentru automatul de potrivire a şirurilor, corespunzător unui model dat P. Algoritmul trebuie sa aibă timpul de execuţie $O(m|\Sigma|)$. (*Indicaţie:* demonstraţi că $\delta(q, a) = \delta(\pi[q], a)$ dacă $q = m$ sau $P[q+1] \neq a$.)

34.5. Algoritmul Boyer-Moore

Dacă modelul P este relativ lung şi alfabetul Σ este destul de mare, atunci algoritmul datorat lui Robert S. Boyer şi J. Strother Moore este, probabil, cel mai eficient pentru potrivirea şirurilor.

POTRIVIRE-BOYER-MOORE(T,P,Σ)

```
 1: n ← lungime[T]
 2: m ← lungime[P]
 3: λ ← CALCUL-FUNCŢIE-ULTIMA-APARIŢIE(P,m,Σ)
 4: γ ← CALCUL-FUNCŢIE-SUFIX-BUN(P,m)
 5: s ← 0
 6: cât timp s ≤ n − m execută
 7:    j ← m
 8:    cât timp j > 0 şi P[j] = T[s + j] execută
 9:       j ← j − 1
10:       dacă j = 0 atunci
11:          tipăreşte "Modelul apare cu deplasamentul" s
12:          s ← s + γ[0]
13:       altfel
14:          s ← s + max(γ[j], j − λ[T[s + j]])
```

În afară de aspectul obscur al lui λ şi γ, acest program este foarte asemănător algoritmului naiv de potrivire a şirurilor. Într-adevăr, presupunem că liniile 3–4 le comentăm şi înlocuim actualizarea lui s din liniile 12–14 cu simple incrementări după cum urmează:

```
12 :   s ← s + 1
13 : altfel
14 :   s ← s + 1
```

Programul modificat funcţionează acum exact ca procedura naivă de potrivire a şirurilor: ciclul **cât timp**, care începe în linia 6, consideră cele $n - m + 1$ deplasamente posibile s, iar ciclul **cât timp**, care începe în linia 8, testează condiţia $P[1..m] = T[s + 1..s + m]$, comparând $P[j]$ cu $T[s + j]$, pentru $j = m, m - 1, ..., 1$. Dacă ciclul se termină cu $j = 0$, atunci a fost determinat un deplasament s corect, şi linia 11 tipăreşte valoarea s. În acest stadiu, singura caracteristică remarcabilă a algoritmului Boyer-Moore este că el compară modelul în sens invers, *de la dreapta spre stânga*, şi, de aceea, creşterea deplasamentului s din liniile 12–14 nu este neapărat 1.

Algoritmul Boyer-Moore include două euristici care permit evitarea multor operaţii pe care le fac algoritmii anteriori de potrivire a şirurilor. Aceste euristici sunt atât de eficiente, încât, adesea, permit algoritmului să omită complet examinarea multor caractere din text. Aceste euristici, cunoscute ca "euristica bazată pe caracterul slab" şi "euristica sufixului bun", sunt ilustrate în figura 34.11. Ele pot fi văzute ca lucrând independent una de alta, în paralel. Când apare o nepotrivire, fiecare euristică propune o valoare cu care poate fi mărit s fără pierderea unui deplasament corect. Algoritmul Boyer-Moore alege cea mai mare valoare şi o adaugă lui s: când se ajunge în linia 13, după o nepotrivire, euristica bazată pe caracterul slab propune incrementarea lui s cu $j - \lambda[T[s + j]]$, iar euristica sufixului bun propune incrementarea lui s cu $\gamma[j]$.

Euristica bazată pe caracterul slab

Când apare o nepotrivire, euristica bazată pe caracterul slab foloseşte informaţia despre poziţia din model în care apare caracterul slab $T[s + j]$ din text (în cazul în care apare) pentru a propune un deplasament nou. În cel mai bun caz, nepotrivirea apare la prima comparaţie ($P[m] \neq T[s + m]$) şi caracterul slab $T[s + m]$ nu apare deloc în model. (Să ne imaginăm căutarea lui a^m în şirul text b^n.) În acest caz, putem mări deplasamentul s cu m deoarece orice deplasament mai mic decât $s + m$ va poziţiona câteva caractere din model peste caracterul slab, provocând o nepotrivire. Dacă cel mai bun caz apare în mod repetat, atunci algoritmul Boyer-Moore examinează numai o fracţiune $1/m$ a textului, deoarece fiecare caracter text examinat produce o nepotrivire în urma căreia s se măreşte cu m. Comportamentul acestui cel mai bun caz ilustrează puterea căutării de la dreapta la stânga spre deosebire de căutarea de la stânga la dreapta.

În continuare, vom vedea cum funcţionează, în general, ***euristica bazată pe caracterul slab***. Presupunem că tocmai am descoperit o nepotrivire: $P[j] \neq T[s+j]$ pentru un j, $1 \leq j \leq m$. Fie k cel mai mare index în intervalul $1 \leq k \leq m$, astfel încât $T[s + j] = P[k]$, dacă există un astfel de k. Altfel, fie $k = 0$. Afirmăm că este corect să îl mărim pe s cu $j - k$. Pentru a demonstra această afirmaţie trebuie să considerăm trei cazuri, aşa cum este ilustrat în figura 34.12.

- $k = 0$: Aşa cum este arătat în figura 34.12(a), caracterul slab $T[s + j]$ nu apare deloc în model şi, deci, putem mări pe s cu j fără a risca pierderea vreunui deplasament corect.
- $k < j$: Aşa cum este arătat în figura 34.12(b), cea mai din dreapta apariţie a caracterului slab în model este la stânga poziţiei j, astfel că $j - k > 0$, şi modelul trebuie deplasat cu $j - k$ caractere spre dreapta. Deci putem mări pe s cu $j - k$ fără a risca pierderea vreunui deplasament corect.
- $k > j$: Aşa cum este arătat în figura 34.12(c), $j - k < 0$ şi deci euristica bazată pe caracterul slab propune decrementarea lui s. Această recomandare va fi ignorată de algoritmul Boyer-Moore deoarece euristica sufixului bun va propune o deplasare spre dreapta în toate cazurile.

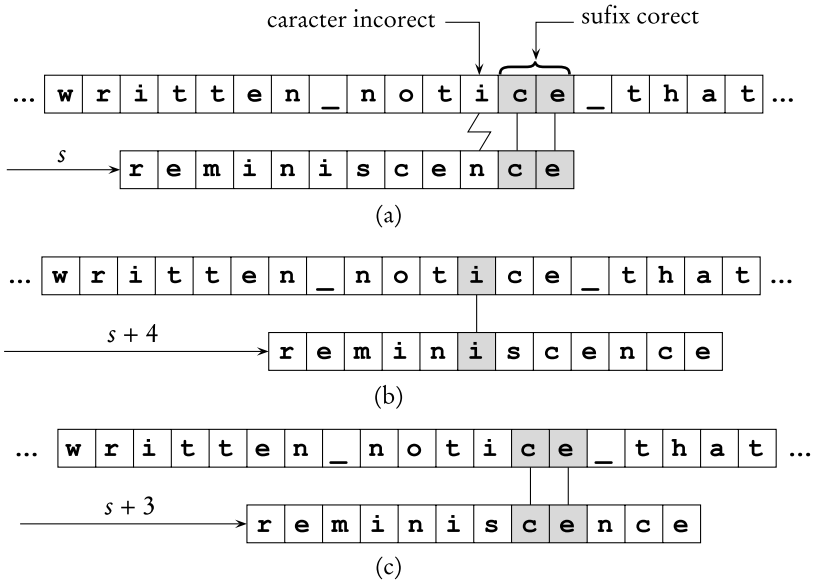


Figura 34.11 O ilustrare a euristicii algoritmului Boyer-Moore. **(a)** Potrivirea modelului `reminiscence` pentru un text, comparând caracterele de la dreapta spre stânga. Deplasamentul s este incorect; deşi "sufixul bun" `ce` al modelului se potriveşte corect pentru caracterele corespunzătoare din text (caracterele potrivite sunt haşurate, în text apare "caracterul slab" `i`, care nu se potriveşte cu caracterul `n` corespunzător din model. **(b)** Euristica bazată pe caracterul slab propune, dacă este posibil, deplasarea modelului spre dreapta, astfel încât să garanteze, pentru caracterul slab din text, că se va potrivi cu cea mai din dreapta apariţie a caracterului slab din model. În acest exemplu, deplasând modelul cu 4 poziţii spre dreapta, caracterul slab `i` din text se potriveşte cu caracterul cel mai din dreapta `i` din model, pe poziţia 6. În cazul în care caracterul slab nu apare în model, acesta poate fi mutat complet peste caracterul slab din text. Dacă cea mai din dreapta apariţie a caracterului slab în model este imediat în dreapta poziţiei caracterului slab curent, atunci această euristică nu face nici o propunere. **(c)** Cu euristica sufixului bun, modelul este deplasat spre dreapta cu cel mai mic deplasament, astfel încât să garanteze că orice caractere din model, care apar în dreptul sufixului bun (găsit în text) se vor potrivi cu toate caracterele sufixului. În exemplu, deplasând modelul cu 3 poziţii spre dreapta, se satisface această condiţie. Deoarece eurisica sufixului bun propune o deplasare cu 3 poziţii, iar euristica bazată pe caracterul slab propune o deplasare mai mare, cu 4 poziţii, algoritmul Boyer-Moore incrementează deplasamentul cu 4.

Următorul program defineşte $\lambda[a]$ ca fiind indicele celei mai din dreapta poziţii din model la care apare caracterul a, oricare ar fi $a \in \Sigma$. Dacă a nu apare în model, atunci $\lambda[a]$ este 0. Numim λ ***funcţia ultimei apariţii*** pentru model. Cu această definiţie, expresia $j - \lambda[T[s+j]]$ din linia 13 a procedurii POTRIVIRE-BOYER-MOORE implementează euristica bazată pe caracterul slab. (Deoarece $j - \lambda[T[s+j]]$ este negativ, dacă cea mai din dreapta apariţie a caracterului slab $T[s+j]$ în model este la dreapta poziţiei j, ne vom baza pe pozitivitatea lui $\gamma[j]$, propus de euristica sufixului bun, pentru a ne asigura că algoritmul face progrese la fiecare pas.)

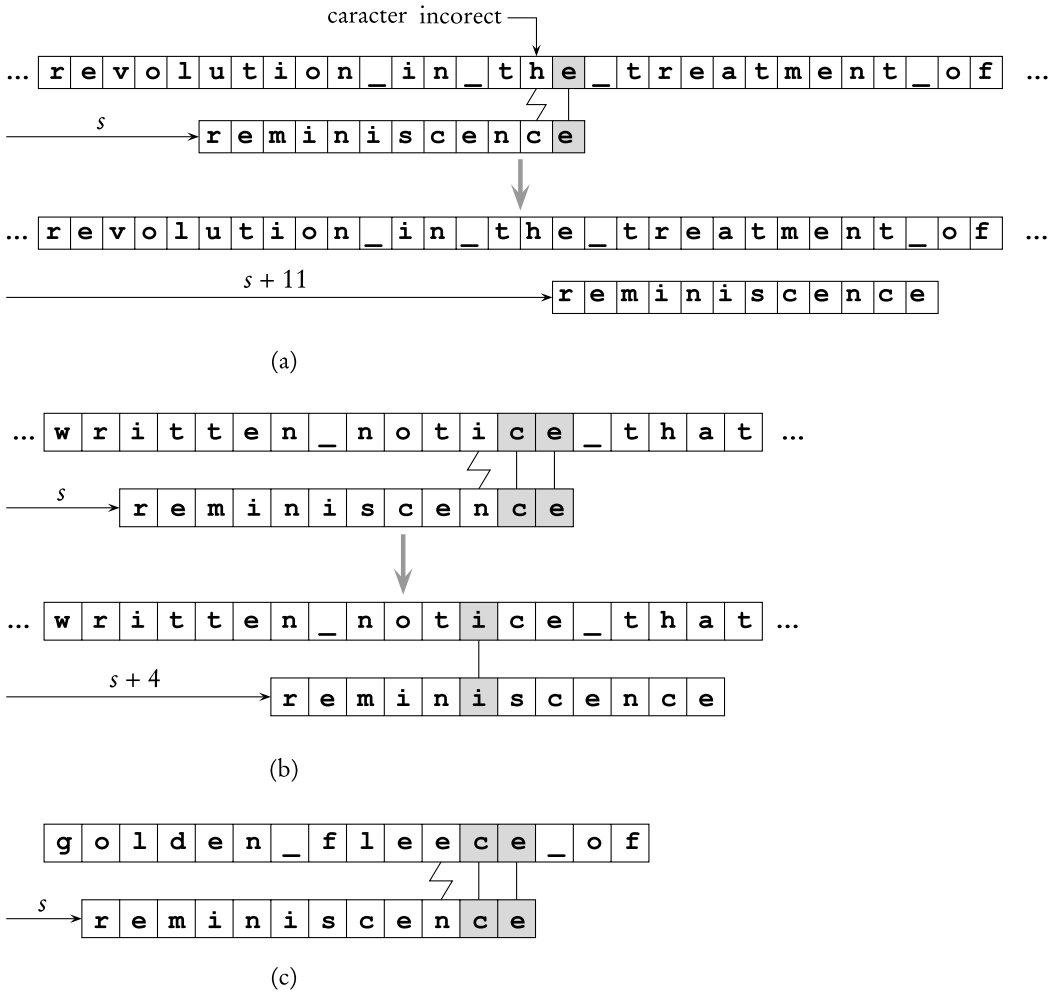


Figura 34.12 Cazul euristicii caracterului slab. **(a)** Caracterul slab `h` nu apare deloc în model şi astfel modelul poate fi avansat cu $j = 11$ caractere până când trece peste caracterul slab. **(b)** Cea mai din dreapta apariţie a caracterului slab în model este la poziţia $k < j$ şi, deci, modelul poate fi avansat cu $j - k$ caractere. Deoarece $j = 10$ şi $k = 6$ pentru caracterul slab `i`, modelul poate fi avansat cu 4 poziţii până când linia lui `i` devine dreaptă. **(c)** Cea mai din dreapta apariţie a caracterului slab în model este la poziţia $k > j$. În acest exemplu, $j = 10$ şi $k = 12$ pentru caracterul slab `e`. Euristica bazată pe caracterul slab propune o deplasare negativă, care este ignorată.

CALCUL-FUNCŢIE-ULTIMA-APARIŢIE(P,m,Σ)
1: **pentru** fiecare caracter $a \in \Sigma$ **execută**
2: $\quad \lambda[a] \leftarrow 0$
3: **pentru** $j \leftarrow 1$, m **execută**
4: $\quad \lambda[P[j]] \leftarrow j$
5: **returnează** λ

Timpul de execuţie al procedurii CALCUL-FUNCŢIE-ULTIMA-APARIŢIE este $O(|\Sigma| + m)$.

Euristica sufixului bun

Pentru şirurile Q şi R definim relaţia $Q \sim R$ (citim "Q este asemenea cu R") astfel: $Q \sqsupset R$ sau $R \sqsupset Q$. Dacă două şiruri sunt asemenea, atunci le putem alinia raportat la cel mai din dreapta caracter care se potriveşte. Relaţia "$\sim$" este simetrică: $Q \sim R$ dacă, şi numai dacă, $R \sim Q$. De asemenea, ca o consecinţă a lemei 34.1, avem că

$$Q \sqsupset R \text{ şi } S \sqsupset R \text{ implică } Q \sim S \tag{34.7}$$

Dacă găsim că $P[j] \neq T[s + j]$, unde $j < m$, atunci ***euristica sufixului bun*** spune că putem mări fără risc s cu

$$\gamma[j] = m - \max\{k : 0 \leq k < m \text{ şi } P[j + 1..m] \sim P_k\}.$$

Deci, $\gamma[j]$ este cel mai mic deplasament cu care poate avansa s fără a determina, la noua aliniere a modelului, apariţia unui caracter slab în "sufixul bun" $T[s + j + 1..s + m]$. Funcţia γ este bine definită pentru orice j deoarece $P[j+1..m] \sim P_0$ pentru orice j: şirul vid este asemenea cu toate şirurile. Numim γ ***funcţia sufixului bun*** pentru modelul P.

Vom arăta acum modul de calcul al funcţiei sufixului bun γ. Observăm, mai întâi, că $\gamma[j] \leq m - \pi[m]$ pentru orice j. Dacă $w = \pi[m]$, atunci $P_w \sqsupset P$ din definiţia lui π. Mai mult, întrucât $P[j+1..m] \sqsupset P$ pentru orice j, avem $P_w \sim P[j+1..m]$ din ecuaţia (34.7). Aşadar, $\gamma[j] \leq m-\pi[m]$ pentru orice j.

Putem acum rescrie definiţia lui γ astfel:

$$\gamma[j] = m - \max\{k : \pi[m] \leq k < m \text{ şi } P[j + 1..m] \sim P_k\}.$$

Condiţia ca $P[j+1..m] \sim P_k$ este îndeplinită dacă fiecare $P[j+1..m] \sqsupset P_k$ sau $P_k \sqsupset P[j + 1..m]$. Dar ultima posibilitate implică $P_k \sqsupset P$ şi, deci, $k \leq \pi[m]$, din definiţia lui π. Această ultimă posibilitate nu poate reduce valoarea lui $\gamma[j]$ sub $m - \pi[m]$. Aşadar, putem rescrie definiţia pe care o vom folosi în continuare pentru γ după cum urmează:

$$\gamma[j] = m - \max(\{\pi[m]\} \cup \{k : \pi[m] < k < m \text{ şi } P[j + 1..m] \sqsupset P_k\}).$$

(A doua mulţime poate fi vidă.) Din definiţie reiese că $\gamma[j] > 0$ pentru orice $j = 1, 2, ..., m$. Aceasta dovedeşte că algoritmul Boyer-Moore face progrese.

Pentru a simplifica expresia pentru γ, definim, în plus, P' reversul modelului P şi π' funcţia prefix corespunzătoare. Acestea sunt, pentru $i = 1, 2, ..., m$, $P'[i] = P[m - i + 1]$ şi $\pi'[t]$ cel mai mare u astfel încât $u < t$ şi $P'_u \sqsupset P'_t$.

Dacă k este cea mai mare valoare posibilă, astfel încât $P[j + 1..m] \sqsupset P_k$, atunci cerem ca

$$\pi'[l] = m - j, \tag{34.8}$$

unde $l = (m-k)+(m-j)$. Pentru a vedea că această cerinţă este bine definită, observăm că $P[j+1..m] \sqsupset P_k$ implică $m-j \leq k$ şi, deci, $l \leq m$. De asemenea, $j < m$ şi $k \leq m$, deci $l \geq 1$. Vom demonstra această cerinţă în continuare. Întrucât $P[j+1..m] \sqsupset P_k$, avem $P'_{m-j} \sqsupset P'_l$. Deci, $\pi'[l] \geq m-j$. Presupunem acum că $p > m-j$, unde $p = \pi'[l]$. Atunci, din definiţia lui π', avem că $P'_p \sqsupset P'_l$ sau, în mod echivalent, $P'[1..p] = P'[l-p+1..l]$. Rescriind această ecuaţie cu P în loc de P', avem $P[m-p+1..m] = P[m-l+1..m-l+p]$. Efectuând substituţia $l = 2m-k-j$, obţinem $P[m-p+1..m] = P[k-m+j+1..k-m+j+p]$ de unde rezultă $P[m-p+1..m] \sqsupset P_{k-m+j+p}$. Din $p > m-j$, avem că $j+1 > m-p+1$, deci $P[j+1..m] \sqsupset P[m-p+1..m]$, rezultă că $P[j+1..m] \sqsupset P_{k-m+j+p}$ din tranzitivitatea relaţiei $\sqsupset$. În final, din $p > m-j$, avem $k' > k$, unde $k' = k-m+j+p$, contrazicând alegerea lui k ca cea mai mare valoare posibilă, astfel încât $P[j+1..m] \sqsupset P_k$. Din această contradicţie, deducem că nu putem avea $p > m-j$, deci $p = m-j$. Cerinţa (34.8) este demonstrată.

Folosind ecuaţia (34.8) şi observând că din $\pi'[l] = m-j$ rezultă $j = m-\pi'[l]$ şi $k = m-l+\pi'[l]$, putem rescrie definiţia noastră pentru γ astfel:

$$\begin{aligned} \gamma[j] &= m - \max(\{\pi[m]\} \cup \{m-l+\pi'[l] : 1 \leq l \leq m \text{ şi } j = m-\pi'[l]\}) \\ &= \min(\{m-\pi[m]\} \cup \{l-\pi'[l] : 1 \leq l \leq m \text{ şi } j = m-\pi'[l]\}). \end{aligned} \tag{34.9}$$

Din nou, a doua mulţime poate fi vidă.

În continuare, vom studia procedura pentru calculul lui γ.

```
CALCUL-FUNCŢIE-SUFIX-BUN(P,m)
 1: π ← CALCUL-FUNCŢIE-PREFIX(P)
 2: P' ← invers(P)
 3: π' ← CALCUL-FUNCŢIE-PREFIX(P')
 4: pentru j ← 0, m execută
 5:    γ[j] ← m − π[m]
 6: pentru l ← 1, m execută
 7:    j ← m − π'[l]
 8:    dacă γ[j] > l − π'[l] atunci
 9:       γ[j] ← l − π'[l]
10: returnează γ
```

Procedura CALCUL-FUNCŢIE-SUFIX-BUN este o implementare simplă a ecuaţiei (34.9). Timpul de execuţie al acesteia este $O(m)$.

În cel mai defavorabil caz, timpul de execuţie pentru algoritmul Boyer-Moore este $O((n-m+1)m + |\Sigma|)$, întrucât CALCUL-FUNCŢIE-ULTIMA-APARIŢIE are timpul de execuţie $O(m+|\Sigma|)$, CALCUL-FUNCŢIE-SUFIX-BUN are timpul de execuţie $O(m)$ şi algoritmul Boyer-Moore (la fel ca algoritmul Rabin-Karp) consumă un timp $O(m)$ validând fiecare deplasament corect s. Cu toate acestea, în practică, algoritmul Boyer-Moore este, adesea, cea mai bună alegere.

Exerciţii

34.5-1 Calculaţi funcţiile λ şi γ pentru modelul $P = \texttt{0101101201}$ şi alfabetul $\Sigma = \{0,1,2\}$.

34.5-2 Daţi exemple pentru a arăta că algoritmul Boyer-Moore poate da performanţe mult mai bune combinând euristica bazată pe caracterul slab şi euristica sufixului bun decât dacă acesta foloseşte numai euristica sufixului bun.

34.5-3 ⋆ O îmbunătăţire, folosită adesea în practică, la procedura de bază Boyer-Moore este înlocuirea funcţiei γ cu funcţia γ' definită astfel:

$$\gamma'[j] = m - \max\{k : \ 0 \le k < m \text{ şi } P[j+1..m] \sim P_k \text{ şi } (k - m + j > 0) \text{ implică } P[j] \ne P[k - m + j])\}.$$

Funcţia γ' garantează că acelaşi model nu va fi potrivit peste textul greşit, asigurând şi că, în sufixul bun, caracterele nu se vor potrivi la un nou deplasament. Arătaţi cum se calculează eficient funcţia γ'.

Probleme

34-1 Potivirea şirurilor bazată pe factori de repetiţie

Notăm cu y^i concatenarea şirului y cu el însuşi de i ori. De exemplu, $(\texttt{ab})^3 = \texttt{ababab}$. Spunem că şirul $x \in \Sigma^*$ are ***factor de repetiţie*** r dacă $x = y^r$ pentru un şir $y \in \Sigma^*$ şi un $r > 0$. Fie $\rho(x)$ cel mai mare r, astfel încât x are factorul de repetiţie r.

a. Daţi un algoritm eficient care are ca date de intrare un model $P[1..m]$ şi calculează $\rho(P_i)$ pentru $i = 1, 2, ..., m$. Care este timpul de execuţie pentru algoritmul pe care l-aţi dat?

b. Pentru orice model $P[1..m]$, definim $\rho^*(P)$ ca $\max_{1 \le i \le m} \rho(P_i)$. Demonstraţi că, dacă modelul P este ales aleator din mulţimea tuturor şirurilor binare de lungime m, atunci valoarea lui $\rho^*(P)$ este $O(1)$.

c. Argumentaţi faptul că următorul algoritm de potrivire a şirurilor găseşte, corect, toate apariţiile modelului P într-un text $T[1..n]$ în timpul $O(\rho^*(P)n + m)$.

```
POTRIVIRE-REPETITIVĂ(P,T)
 1: m ← lungime[P]
 2: n ← lungime[T]
 3: k ← 1 + ρ*(P)
 4: q ← 0
 5: s ← 0
 6: cât timp s ≤ n − m execută
 7:    dacă T[s + q + 1] = P[q + 1] atunci
 8:       q ← q + 1
 9:       dacă q = m atunci
10:          tipăreşte "Modelul apare cu deplasamentul" s
11:    dacă q = m sau T[s + q + 1] ≠ P[q + 1] atunci
12:       s ← s + max(1, ⌈q/k⌉)
13:       q ← 0
```

Acest algoritm este datorat lui Galil şi Seiferas. Extinzând mult aceste idei ei obţin un algoritm de potrivire a şirurilor de timp liniar. Acesta foloseşte numai $O(1)$ spaţiu de memorie în afară de ceea ce este necesar pentru P şi T.

34-2 Potrivirea şirurilor în paralel

Analizăm problema de potrivire a şirurilor pe un calculator paralel. Presupunem că, pentru un model dat, avem un automat de potrivire a şirurilor M cu mulţimea de stări Q. Fie ϕ funcţia de stare finală pentru M. Textul de intrare este $T[1..n]$. Dorim să calculăm $\phi(T_i)$ pentru $i = 1, 2, ..., n$; adică, dorim să calculăm starea finală pentru fiecare prefix. Planul nostru este să folosim calculul de prefix paralel descris în secţiunea 30.1.2.

Pentru fiecare şir de intrare x, definim funcţia $\delta_x : Q \to Q$, astfel încât, dacă automatul M porneşte în starea q şi citeşte intrarea x, atunci M se termină în starea $\delta_x(q)$.

a. Demonstraţi că $\delta_y \circ \delta_x = \delta_{xy}$, unde cu $\circ$ am notat funcţia de compunere:

$$(\delta_y \circ \delta_x)(q) = \delta_y(\delta_x(q)).$$

b. Arătaţi că $\circ$ este o operaţie asociativă.

c. Arătaţi că δ_{xy} poate fi calculată din reprezentările tabelare ale lui δ_x şi δ_y în timpul $O(1)$ pe o maşină CREW PRAM. Analizaţi în funcţie de $|Q|$ câte procesoare sunt necesare.

d. Demonstraţi că $\phi(T_i) = \delta_{T_i}(q_0)$, unde q_0 este starea de start pentru M.

e. Arătaţi cum se găsesc toate potrivirile unui model într-un text de lungime n în timpul $O(\lg n)$ pe o maşină CREW PRAM. Presupunem că modelul este dat corespunzător în forma automatului de potrivire a şirurilor.

Note bibliografice

Relaţia dintre potrivirea şirurilor şi teoria automatelor finite este discutată de Aho, Hopcroft şi Ullman [4]. Algoritmul Knuth-Morris-Pratt [125] a fost inventat independent de către Knuth şi Pratt şi de către Morris; ei şi-au publicat rezultatele reunite. Algoritmul Rabin-Karp a fost propus de Rabin şi Karp [117], iar algoritmul Boyer-Moore este datorat lui Boyer şi Moore [32]. Galil şi Seiferas [78] dau un algoritm deterministic interesant de potrivire a şirurilor în timp liniar care foloseşte doar $O(1)$ spaţiu de memorie în afara celui necesar pentru memorarea modelului şi textului.

35 Geometrie computaţională

Geometria computaţională este ramura ştiinţei informaticii care studiază algoritmi pentru rezolvarea problemelor geometrice. Geometria computaţională are aplicaţii în ingineria modernă, în matematică şi în alte domenii cum ar fi grafica pe calculator, robotica, proiectarea VLSI, proiectarea asistată de calculator şi statistica. Datele de intrare, pentru o problemă de geometrie computaţională, sunt, de obicei, o descriere a unei mulţimi de obiecte geometrice, ca o mulţime de puncte, o mulţime de segmente de dreaptă sau o mulţime de vârfuri ale unui poligon, ordonate în sens trigonometric. Ieşirea este, adesea, un răspuns la o întrebare despre obiecte, cum ar fi intersectarea unor drepte sau un nou obiect geometric, de exemplu, învelitoarea convexă a mulţimii de puncte (cel mai mic poligon convex care le include).

În acest capitol, urmărim câţiva algoritmi de geometrie computaţională în două dimensiuni, adică în plan. Fiecare obiect de intrare este reprezentat ca o mulţime de puncte $\{p_i\}$, unde $p_i = (x_i, y_i)$ şi $x_i, y_i \in \mathbb{R}$. De exemplu, un poligon P cu n vârfuri este reprezentat prin secvenţa $\langle p_0, p_1, p_2, ..., p_{n-1} \rangle$ a vârfurilor sale în ordinea apariţiei lor în P. De asemenea, geometria computaţională se poate face în trei dimensiuni şi chiar în spaţii dimensionale mai mari, dar astfel de probleme şi soluţiile lor pot fi vizualizate foarte greu. Chiar şi în două dimensiuni, putem vedea exemple bune de tehnici de geometrie computaţională.

În secţiunea 35.1 prezentăm răspunsuri corecte şi eficiente la întrebări simple despre segmente de dreaptă: dacă un segment este în sensul acelor de ceasornic sau în sens trigonometric faţă de un alt segment cu care are un capăt comun, în ce mod ne întoarcem când parcurgem două segmente de dreaptă adiacente, dacă două segmente de dreaptă se intersectează. În secţiunea 35.2 se prezintă o tehnică numită "baleiere" pe care o vom folosi pentru a dezvolta un algoritm de timp $O(n \lg n)$ care determină dacă există intersecţii pentru o mulţime de n segmente de dreaptă. În secţiunea 35.3 se prezintă doi algoritmi de "baleiaj rotaţional" care calculează învelitoarea convexă pentru o mulţime de n puncte (cel mai mic poligon convex care le include): scanarea lui Graham, care se execută în timpul $O(n \lg n)$, şi potrivirea lui Jarvis, care are timpul de execuţie $O(nh)$, unde h este numărul de vârfuri ale învelitorii convexe. În final, în secţiunea 35.4 se prezintă un algoritm divide şi stăpâneşte de timp $O(n \lg n)$ pentru determinarea celei mai apropiate perechi de puncte dintr-o mulţime de n puncte din plan.

35.1. Proprietăţile segmentului de dreaptă

În acest capitol, majoritatea algoritmilor de geometrie computaţională vor cere răspunsuri la întrebări despre proprietăţile segmentelor de dreaptă. O ***combinare convexă*** a două puncte distincte $p_1 = (x_1, y_1)$ şi $p_2 = (x_2, y_2)$ este un punct $p_3 = (x_3, y_3)$, astfel încât pentru un α oarecare, în intervalul $0 \leq \alpha \leq 1$, avem $x_3 = \alpha x_1 + (1 - \alpha)x_2$ şi $y_3 = \alpha y_1 + (1 - \alpha)y_2$. De asemenea, vom scrie că $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Intuitiv, p_3 este pe dreapta care trece prin punctele p_1 şi p_2 şi este în afară sau între p_1 şi p_2 pe această dreaptă. Pentru două puncte p_1 şi p_2 date, ***segmentul de dreaptă*** $\overline{p_1 p_2}$ este mulţimea combinărilor convexe ale lui p_1 şi p_2. Numim p_1 şi p_2 ***capetele*** segmentului $\overline{p_1 p_2}$. Câteodată, contează ordinea punctelor p_1 şi p_2, atunci vorbim despre ***segment orientat*** $\overrightarrow{p_1 p_2}$. Dacă p_1 este ***originea*** $(0, 0)$, atunci putem trata segmentul

orientat $\overrightarrow{p_1p_2}$ ca fiind ***vectorul*** p_2.

În această secţiune, vom căuta răspunsuri la următoarele întrebări:

1. Dându-se două segmente orientate $\overrightarrow{p_0p_1}$ şi $\overrightarrow{p_0p_2}$ ţinând seama de capătul lor comun p_0 este $\overrightarrow{p_0p_1}$ în sensul acelor de ceasornic de la $\overrightarrow{p_0p_2}$?

2. Fiind date două segmente de dreaptă $\overline{p_1p_2}$ şi $\overline{p_2p_3}$, dacă parcurgem $\overline{p_1p_2}$ şi apoi $\overline{p_2p_3}$, facem o întoarcere la stânga în punctul p_2?

3. Se intersectează segmentele de dreaptă $\overline{p_1p_2}$ şi $\overline{p_3p_4}$?

Nu există restricţii pentru punctele date.

Putem răspunde la fiecare întrebare în timpul $O(1)$, ceea ce ar trebui să nu ne surprindă deoarece dimensiunea datelor de intrare, pentru fiecare întrebare, este $O(1)$. Mai mult, metodele noastre folosesc doar adunări, scăderi, înmulţiri şi comparaţii. Nu avem nevoie nici de împărţiri, nici de funcţii trigonometrice, amândouă pot necesita calcule costisitoare şi sunt predispuse problemelor generate de erori de rotunjire. De exemplu, o metodă "simplă" pentru a determina dacă două segmente se intersectează – calculăm ecuaţia dreptei în forma $y = mx + b$ pentru fiecare segment (m este panta şi b este intersecţia dreptei cu Oy), găsim punctul de intersecţie al celor două drepte şi verificăm dacă el aparţine ambelor segmente – folosim împărţirea pentru a determina punctul de intersecţie. Când segmentele sunt aproape paralele, această metodă este foarte sensibilă la precizia operaţiei de împărţire pe un calculator real. În această secţiune, metoda care evită operaţia de împărţire este mult mai exactă.

Produse încrucişate

Calcularea produsului încrucişat este esenţa metodelor noastre. Considerăm vectorii p_1 şi p_2, prezentaţi în figura 35.1(a). ***Produsul încrucişat*** $p_1 \times p_2$ poate fi interpretat ca fiind aria cu semn a paralelogramului format din punctele $(0,0), p_1, p_2$ şi $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. O definiţie echivalentă, dar mult mai utilă, pentru produsul încrucişat este dată de determinantul matricei:[1]

$$p_1 \times p_2 \quad = \quad \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1y_2 - x_2y_1 = -p_2 \times p_1.$$

Dacă $p_1 \times p_2$ este pozitiv, atunci p_1 este în sensul acelor de ceasornic faţă de p_2 ţinând cont de originea $(0,0)$; dacă acest produs încrucişat este negativ, atunci p_1 este în sens trigonometric faţă de p_2. În figura 35.1(b) se prezintă regiunile relative la vectorul p, în sensul acelor de ceasornic şi în sensul trigonometric. Condiţia limită apare dacă produsul încrucişat este zero; în acest caz, vectorii sunt ***coliniari***, orientaţi în aceeaşi direcţie sau în direcţii opuse.

Pentru a determina dacă un segment orientat $\overrightarrow{p_0p_1}$ este în sensul acelor de ceasornic faţă de segmentul orientat $\overrightarrow{p_0p_2}$ în raport cu punctul lor comun p_0, facem o simplă translaţie cu scopul de a folosi p_0 ca origine. Adică, notăm cu $p_1 - p_0$ vectorul $p_1' = (x_1', y_1')$, unde $x_1' = x_1 - x_0$ şi $y_1' = y_1 - y_0$. Definim în mod analog $p_2 - p_0$. Calculăm apoi produsul încrucişat

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0).$$

[1]De fapt, produsul încrucişat este un concept tri-dimensional. Este un vector perpendicular pe p_1 şi pe p_2, conform "regulii mâinii drepte", şi al cărui modul este $|x_1y_2 - x_2y_1|$. În acest capitol. se va dovedi convenabilă tratarea produsului încrucişat ca simpla valoare $x_1y_2 - x_2y_1$.

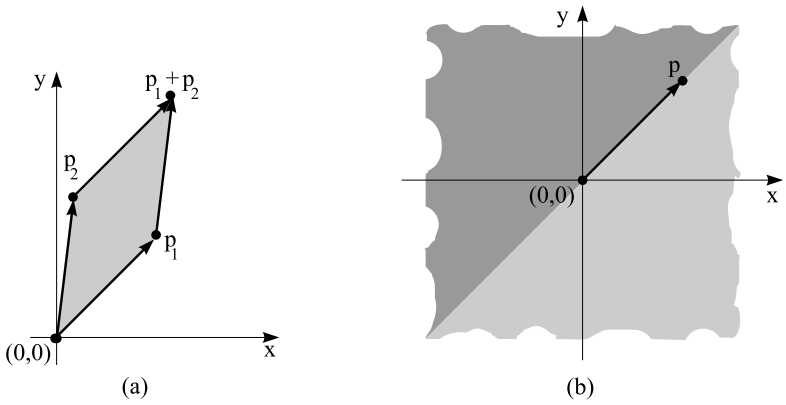


Figura 35.1 **(a)** Produsul încrucişat al vectorilor p_1 şi p_2 este aria cu semn a paralelogramului. **(b)** Regiunea albă conţine vectorii care sunt în sensul acelor de ceasornic faţă de p. Regiunea haşurată conţine vectorii care sunt în sens trigonometric faţă de p.

Dacă acest produs încrucişat este pozitiv, atunci $\overrightarrow{p_0p_1}$ este în sensul acelor de ceasornic faţă de $\overrightarrow{p_0p_2}$; dacă este negativ, atunci este în sens trigonometric.

Determinarea dacă segmente consecutive se întorc la stânga sau la dreapta

Următoarea întrebare este dacă două segmente de dreaptă consecutive $\overline{p_0p_1}$ şi $\overline{p_1p_2}$ se întorc la stânga sau la dreapta în punctul p_1. Adică, dorim o metodă pentru determinarea direcţiei în care se întoarce un unghi dat $\angle p_0p_1p_2$. Produsul încrucişat ne permite să răspundem la această întrebare fără să calculăm unghiul. Aşa cum se poate vedea în figura 35.2, verificăm dacă segmentul orientat $\overrightarrow{p_0p_2}$ este în sensul acelor de ceasornic sau în sens trigonometric, relativ la segmentul orientat $\overrightarrow{p_0p_1}$. Pentru această verificare, calculăm produsul încrucişat $(p_2-p_0)\times(p_1-p_0)$. Dacă semnul acestui produs încrucişat este negativ, atunci $\overrightarrow{p_0p_2}$ este în sens trigonometric faţă de $\overrightarrow{p_0p_1}$ şi, astfel, facem o întoarcere la stânga în p_1. Un produs încrucişat pozitiv, indică o orientare în sensul acelor de ceasornic şi o întoarcere la dreapta. Un produs încrucişat 0 înseamnă că punctele p_0, p_1 şi p_2 sunt coliniare.

Determinarea faptului dacă două segmente de dreaptă se intersectează

Pentru a determina dacă două segmente de dreaptă se intersectează, vom proceda în două etape. Prima etapă este ***respingerea rapidă***: segmentele de dreaptă nu se pot intersecta dacă dreptunghiurile de mărginire nu se intersectează. ***Dreptunghiul de mărginire*** al unei figuri geometrice este cel mai mic dreptunghi care conţine figura şi ale cărui segmente sunt paralele cu axa-x şi axa-y. Dreptunghiul de mărginire al segmentului de dreaptă $\overline{p_1p_2}$ este reprezentat prin dreptunghiul $(\widehat{p_1}, \widehat{p_2})$ cu punctul stânga jos $\widehat{p_1} = (\widehat{x}_1, \widehat{y}_1)$ şi punctul dreapta sus $\widehat{p_2} = (\widehat{x}_2, \widehat{y}_2)$, unde $\widehat{x}_1 = \min(x_1, x_2), \widehat{y}_1 = \min(y_1, y_2), \widehat{x}_2 = \max(x_1, x_2)$ şi $\widehat{y}_2 = \max(y_1, y_2)$. Două dreptunghiuri reprezentate prin punctele din stânga jos şi din dreapta sus $(\widehat{p_1}, \widehat{p_2})$ şi $(\widehat{p_3}, \widehat{p_4})$ se

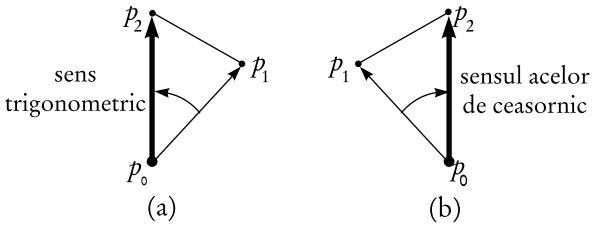


Figura 35.2 Folosirea produsului încrucişat pentru a determina cum se întorc în punctul p_1 segmentele de dreaptă consecutive $\overline{p_0p_1}$ şi $\overline{p_1p_2}$. Verificăm dacă segmentul orientat $\overrightarrow{p_0p_2}$ este în sensul acelor de ceasornic sau în sens trigonometric relativ la segmentul orientat $\overrightarrow{p_0p_1}$. **(a)** Dacă este în sens trigonometric, punctele fac o întoarcere la stânga. **(b)** Dacă este în sensul acelor de ceasornic, ele fac o întoarcere la dreapta.

intersectează dacă, şi numai dacă, este adevărată conjuncţia:

$$(\widehat{x}_2 \geq \widehat{x}_3) \wedge (\widehat{x}_4 \geq \widehat{x}_1) \wedge (\widehat{y}_2 \geq \widehat{y}_3) \wedge (\widehat{y}_4 \geq \widehat{y}_1).$$

Dreptunghiurile trebuie să se intersecteze pe ambele laturi. Primele două comparaţii de mai sus determină dacă dreptunghiurile se intersectează în x; ultimele două comparaţii determină dacă dreptunghiurile se intersectează în y.

A doua etapă în determinarea faptului dacă două segmente de dreaptă se intersectează decide dacă fiecare segment "întretaie" dreapta care conţine celălalt segment. Un segment $\overline{p_1p_2}$ ***întretaie*** o dreaptă dacă punctul p_1 se află de o parte a dreptei, iar p_2 se află de cealaltă parte a ei. Dacă p_1 şi p_2 se află pe dreaptă, atunci spunem că segmentul întretaie dreapta. Două segmente de dreaptă se intersectează dacă, şi numai dacă, ele trec de testul de respingere rapidă şi fiecare segment întretaie dreapta care conţine celălalt segment.

Putem folosi metoda produsului încrucişat pentru a determina dacă segmentul de dreaptă $\overline{p_3p_4}$ întretaie dreapta care conţine punctele p_1 şi p_2. Ideea, aşa cum am arătat în figura 35.3(a) şi (b), este să determinăm dacă segmentele orientate $\overrightarrow{p_1p_3}$ şi $\overrightarrow{p_1p_4}$ au orientări opuse relativ la $\overrightarrow{p_1p_2}$. În acest caz, segmentul întretaie dreapta. Amintim că putem determina orientările relative cu ajutorul produselor încrucişate, verificând doar dacă semnele produselor încrucişate $(p_3 - p_1) \times (p_2 - p_1)$ şi $(p_4 - p_1) \times (p_2 - p_1)$ sunt diferite. O condiţie limită apare dacă oricare dintre produsele încrucişate este zero. În acest caz, fie p_3, fie p_4 se află pe dreapta care conţine segmentul $\overline{p_1p_2}$. Deoarece cele două segmente au trecut deja testul de respingere rapidă, unul dintre punctele p_3 sau p_4 trebuie să se afle, de fapt, pe segmentul $\overline{p_1p_2}$. Două astfel de situaţii sunt ilustrate în figura 35.3(c) şi (d). Cazul în care cele două segmente sunt coliniare, dar nu se intersectează, ilustrat în figura 35.3(e), este eliminat de testul de respingere rapidă. O ultimă condiţie limită apare dacă unul dintre cele două segmente are lungimea zero, altfel spus, capetele segmentului coincid. Dacă ambele segmente au lungimea zero, atunci testul de respingere rapidă este suficient. Dacă doar un segment, să zicem $\overline{p_3p_4}$, are lungimea zero, atunci segmentele se intersectează dacă, şi numai dacă, produsul încrucişat $(p_3 - p_1) \times (p_2 - p_1)$ este zero.

Alte aplicaţii ale produselor încrucişate

În ultima secţiune a acestui capitol se vor prezenta alte utilizări ale produselor încrucişate. În secţiunea 35.3 vom avea nevoie să ordonăm o mulţime de puncte în raport cu unghiurile lor

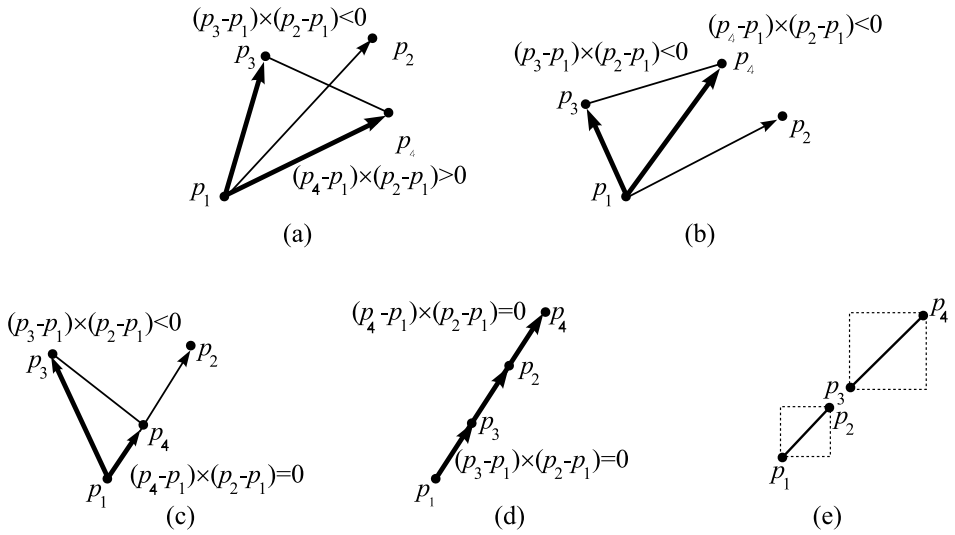


Figura 35.3 Determinarea faptului dacă segmentul de dreaptă $\overline{p_3p_4}$ întretaie dreapta ce conţine segmentul $\overline{p_1p_2}$. **(a)** Dacă acesta întretaie dreapta, atunci semnele produselor încrucişate $(p_3 - p_1) \times (p_2 - p_1)$ şi $(p_4 - p_1) \times (p_2 - p_1)$ diferă. **(b)** Dacă acesta nu întretaie dreapta, atunci produsele sunt de acelaşi semn. **(c)-(d)** Cazurile limită în care cel puţin unul dintre produsele încrucişate este zero şi segmentul întretaie dreapta. **(e)** Cazul limită în care segmentele sunt coliniare, dar nu se intersectează. Ambele produse încrucişate sunt zero, dar ele nu pot fi calculate prin algoritmul nostru deoarece segmentele nu trec testul de respingere rapidă – dreptunghiurile lor de mărginire nu se intersectează.

polare şi o origine dată. Aşa cum se cere să arătaţi în exerciţiul 35.1-2, produsele încrucişate pot fi folosite pentru comparările din procedura de sortare. În secţiunea 35.2, vom folosi arbori roşu-negru pentru a păstra ordonarea verticală a unei mulţimi de segmente de dreaptă care nu se intersectează. Decât să păstrăm valorile cheie explicit, preferăm să înlocuim fiecare comparaţie de cheie, din codul arborilor roşu-negru, cu un calcul de produs încrucişat pentru a determina care dintre două segmente, care intersectează o dreaptă verticală dată, este mai sus.

Exerciţii

35.1-1 Demonstraţi că, dacă $p_1 \times p_2$ este pozitiv, atunci vectorul p_1 este în sensul acelor de ceasornic faţă de vectorul p_2 în raport cu originea $(0, 0)$ şi că, dacă acest produs este negativ, atunci vectorul p_1 este în sens trigonometric faţă de vectorul p_2.

35.1-2 Scrieţi, în pseudocod, o procedură pentru sortarea unei secvenţe $\langle p_1, p_2, ..., p_n \rangle$ de n puncte specificate prin unghiurile lor polare, în raport cu un punct de origine p_0 dat. Procedura trebuie să se execute într-un timp $O(n \lg n)$ şi să folosească produse încrucişate pentru a compara unghiuri.

35.1-3 Arătaţi cum se determină, într-un timp $O(n^2 \lg n)$, dacă oricare trei puncte dintr-o mulţime de n puncte sunt coliniare.

35.1-4 Profesorul Amundsen propune următoarea metodă pentru a determina dacă secvenţa de n puncte $\langle p_0, p_1, ..., p_{n-1}\rangle$ reprezintă vârfurile consecutive ale unui poligon convex. (Vezi secţiunea 16.4 pentru definiţiile relative la poligoane.) Dacă mulţimea $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, ..., n-1\}$, unde adunarea de indici se face modulo n, nu conţine nici întoarceri la stânga, şi nici întoarceri la dreapta, atunci ieşirea este "da"; altfel ieşirea este "nu". Arătaţi că, deşi această metodă are un timp de execuţie liniar, ea nu conduce întotdeauna la rezultatul corect. Modificaţi metoda profesorului astfel încât să producă, întotdeauna, răspunsul corect într-un timp liniar.

35.1-5 Fiind dat un punct $p_0 = (x_0, y_0)$, ***raza orizontală spre dreapta***, din punctul p_0, este mulţimea punctelor $\{p_i = (x_i, y_i) : x_i \geq x_0$ şi $y_i = y_0\}$, adică mulţimea punctelor din dreapta lui p_0 inclusiv p_0. Fiind dată o rază orizontală spre dreapta din p_0, arătaţi cum se determină într-un timp $O(1)$, dacă aceasta se intersectează cu un segment de dreaptă $\overline{p_1 p_2}$. Realizaţi aceasta reducând problema la a determina dacă două segemente de dreaptă se intersectează.

35.1-6 O metodă pentru a determina dacă un punct p_0 se află în interiorul unui poligon P, nu neapărat convex, este să verificăm dacă fiecare rază din p_0 intersectează frontiera lui P de un număr impar de ori, iar punctul p_0 nu este pe frontiera lui P. Arătaţi cum se calculează, într-un timp $\Theta(n)$, dacă un punct p_0 este în interiorul unui poligon P cu n vârfuri. (*Indicaţie:* Folosiţi exerciţiul 35.1-5. Asiguraţi-vă că algoritmul este corect când raza intersectează frontiera poligonului într-un vârf şi când raza se suprapune peste o latură a poligonului.)

35.1-7 Arătaţi cum se calculează aria unui poligon cu n vârfuri, nu neapărat convex, într-un timp $\Theta(n)$.

35.2. Determinarea cazului în care oricare două segmente se intersectează

Această secţiune prezintă un algoritm pentru a determina dacă oricare două segmente de dreaptă, dintr-o mulţime de segmente, se intersectează. Algoritmul foloseşte o tehnică cunoscută sub numele de "baleiere" care este comună multor algoritmi de geometrie computaţională. Mai mult, aşa cum este arătat în exerciţiile de la sfârşitul secţiunii, acest algoritm sau simple variaţiuni ale lui pot fi folosite pentru a rezolva şi alte probleme de geometrie computaţională.

Timpul de execuţie al algoritmului este $O(n \lg n)$, unde n este numărul de segmente date. Algoritmul determină numai dacă două drepte se intersectează sau nu; nu afişează toate intersecţiile. (În exerciţiul 35.2-1, pentru a găsi *toate* intersecţiile într-o mulţime de n segmente de dreaptă, timpul de execuţie, în cel mai defavorabil caz, este $\Omega(n^2)$.

În ***baleiere***, o ***dreaptă de baleiere*** verticală, imaginară, traversează mulţimea de obiecte geometrice dată, de obicei, de la stânga la dreapta. Dimensiunea x a spaţiului în care dreapta de baleiere se deplasează, este tratată ca dimensiunea timpului. Baleierea oferă o metodă pentru ordonarea obiectelor geometrice, de obicei, plasându-le într-o structură de date dinamică, şi pentru obţinerea relaţiilor dintre ele. În această secţiune, algoritmul intersecţie-segment-de-dreaptă consideră toate capetele segmentelor de dreaptă ordonate de la stânga la dreapta şi verifică dacă există o intersecţie de fiecare dată când întâlneşte un capăt de segment.

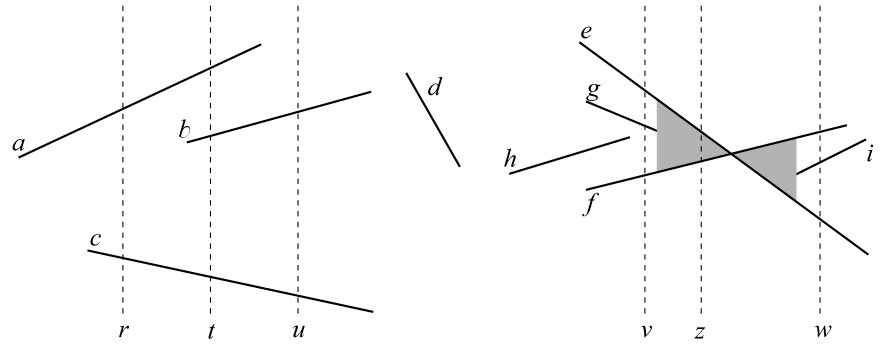


Figura 35.4 Ordinea segmentelor de dreaptă pentru diferite drepte de baleiere verticale. **(a)** Avem $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ şi $b >_u c$. Segmentul d nu este comparabil cu segmentele din figură. **(b)** La momentul intersecţiei segmentului e cu segmentul f, ordinea lor se inversează: avem $e >_v f$, dar $f >_w e$. Orice dreaptă de baleiere (de exemplu z) care traversează regiunea haşurată are în ordinea ei totală segmentele e şi f consecutive.

Ordonarea segmentelor

Dacă presupunem că nu există segmente verticale, atunci orice segment de intrare care intersectează o dreaptă de baleiere verticală dată, o face într-un singur punct. Astfel, putem ordona segmentele care intersectează o dreaptă de baleiere verticală după coordonatele y ale punctelor de intersecţie.

Mai exact, considerăm două segmente s_1 şi s_2 care nu se intersectează. Spunem că aceste segmente sunt ***comparabile*** după x dacă dreapta de baleiere verticală cu coordonata orizontală x le intersectează pe amândouă. Spunem că s_1 este ***deasupra*** lui s_2 după x şi scriem $s_1 >_x s_2$, dacă s_1 şi s_2 sunt comparabile după x şi intersecţia lui s_1 cu dreapta de baleiere din x se află deasupra intersecţiei lui s_2 cu aceeaşi dreaptă de baleiere. De exemplu, în figura 35.4(a) avem relaţiile $a >_r c$, $a >_t b$, $b >_t c$, $a >_t c$ şi $b >_u c$. Segmentul d nu este comparabil cu nici un alt segment.

Pentru orice x dat, relaţia "$>_x$" este o ordine totală (vezi secţiunea 5.2) de segmente care intersectează dreapta de baleiere din x. Pentru valori diferite ale lui x, ordinea poate fi diferită, deşi segmentele intră şi ies din ordonare. Un segment intră în ordonare când capătul stâng este întâlnit de baleiere şi iese din ordonare când este întâlnit capătul drept.

Ce se întâmplă atunci când dreapta de baleiere trece prin intersecţia a două segmente? Aşa cum se vede în figura 35.4(b) este inversată poziţia lor în ordinea totală. Dreptele de baleiere v şi w sunt în dreapta, respectiv, stânga punctului de intersecţie a segmentelor e şi f, şi avem $e >_v f$ dar $f >_w e$. Subliniem aceasta pentru că presupunem că nu există trei segmente care să se intersecteze în acelaşi punct, trebuie să existe vreo dreaptă de baleiere verticală x care intersectează segmentele e şi f şi pentru care ele sunt *consecutive* în ordinea totală $>_x$. Orice dreaptă de baleiere care trece prin regiunea haşurată din figura 35.4(b), cum ar fi z, are segmentele e şi f consecutive în ordinea totală.

Deplasarea dreptei de baleiere

Algoritmii de baleiere, de obicei, gestionează două mulţimi de date:

1. ***Starea liniei de baleiere*** dă relaţia dintre obiectele intersectate de linia de baleiere.

2. ***Lista punct-eveniment*** este o secvenţă de coordonate-x, ordonate de la stânga la dreapta, care definesc poziţiile de oprire ale dreptei de baleiere. Numim fiecare astfel de poziţie de oprire ***punct eveniment***. Numai în punctele eveniment, se întâlnesc modificări ale stării liniei de baleiere.

Pentru unii algoritmi (de exemplu, algoritmul cerut în exerciţiul 35.2-7), lista punct-eveniment este determinată dinamic în timpul execuţiei algoritmului. Cu toate acestea, algoritmul pe care îl avem la îndemână determină, static, punctele eveniment, bazându-se doar pe simple proprietăţi ale datelor de intrare. În particular, fiecare capăt de segment este un punct eveniment. Sortăm capetele de segment prin creşterea coordonatei-x şi procedăm de la stânga la dreapta. Când întâlnim un capăt stâng de segment, inserăm segmentul în stările dreptei de baleiere şi, când întâlnim un capăt drept de segment, ştergem segmentul din stările dreptei de baleiere. De fiecare dată când două segmente devin consecutive în ordinea totală, verificăm dacă ele se intersectează.

Stările dreptei de baleiere sunt o ordine totală T, pentru care avem nevoie de următoarele operaţii:

- INSEREAZĂ(T,s): inserează segmentul s în T.
- ŞTERGE(T,s): şterge segmentul s din T.
- DEASUPRA(T,s): returnează segmentul care este imediat deasupra lui s în T.
- DEDESUBT(T,s): returnează segmentul care este imediat dedesubtul lui s în T.

Dacă există n segmente în mulţimea de intrare, folosind arborii roşu-negru, putem realiza fiecare dintre operaţiile deasupra în timpul $O(\lg n)$. Reamintim că, în capitolul 14, operaţiile arbore-roşu-negru necesită compararea cheilor. Putem înlocui compararea cheilor cu compararea produs încrucişat care determină ordinea relativă a două segmente (vezi exerciţiul 35.2-2).

Algoritm pseudocod pentru intersecţia segmentelor

Algoritmul următor primeşte ca date de intrare o mulţime S de n segmente de dreaptă. Acesta returnează valoarea logică ADEVĂRAT dacă orice pereche de segmente din S se intersectează, iar altfel returnează FALS. Ordinea totală T este implementată printr-un arbore-roşu-negru.

Execuţia algoritmului este ilustrată în figura 35.5. În linia 1 este iniţializată ordinea totală cu mulţimea vidă. Linia 2 determină lista punctelor eveniment prin ordonarea de la stânga la dreapta a celor $2n$ capete de segment, eliminând egalităţile prin punerea capetelor din stânga înaintea capetelor din dreapta.

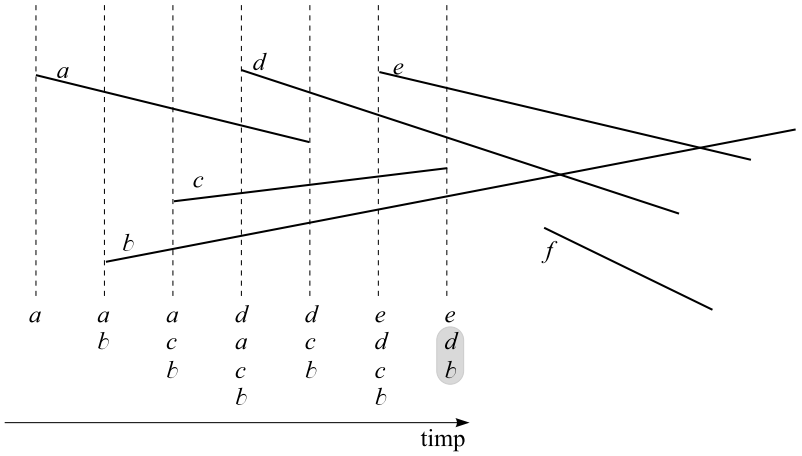


Figura 35.5 Execuţia algoritmului INTERSECŢIA-ORICĂROR-SEGMENTE. Fiecare dreaptă punctată reprezintă o dreaptă de baleiere într-un punct eveniment. Ordinea numelor de segment de sub fiecare dreaptă de baleiere este ordinea totală T la sfârşitul ciclului **pentru** în care este tratat punctul eveniment corespunzător. Intersecţia segmentelor d şi b este găsită când este detectat segmentul c.

INTERSECŢIA-ORICĂROR-SEGMENTE(S)
1: $T \leftarrow \emptyset$
2: sortează capetele segmentelor din S de la stânga la dreapta eliminând egalităţile prin punerea capetelor stânga înaintea capetelor dreapta
3: **pentru** fiecare punct p din lista ordonată de capete **execută**
4: **dacă** p este capătul stâng al segmentului s **atunci**
5: INSEREAZĂ(T,s)
6: **dacă** (DEASUPRA(T,s) există şi intersectează pe s) sau (DEDESUBT(T,s) există şi intersectează pe s) **atunci**
7: **returnează** ADEVĂRAT
8: **dacă** p este capătul drept al segmentului s **atunci**
9: **dacă** există ambele segmente DEASUPRA(T,s) şi DEDESUBT(T,s) şi DEASUPRA(T,s) intersectează DEDESUBT(T,s) **atunci**
10: **returnează** ADEVĂRAT
11: ŞTERGE(T,s)
12: **returnează** FALS

Fiecare iteraţie a ciclului **pentru** din liniile 3–11 tratează un punct eveniment p. Dacă p este capătul din stânga al segmentului s, atunci linia 5 adaugă s la ordinea totală şi liniile 6–7 returnează ADEVĂRAT dacă s intersectează ambele segmente ce îi sunt consecutive în ordinea totală definită de linia de baleiere ce trece prin p. (O condiţie limită apare dacă p se află pe un alt segment s'. În acest caz, este necesar doar ca s şi s' să fie consecutive în T.) Dacă p este capătul din dreapta al segmentului s, atunci s este şters din ordinea totală. Liniile 9–10 returnează ADEVĂRAT dacă există o intersecţie între segmentele ce îi sunt consecutive lui s în ordinea totală definită de dreapta de baleiere ce trece prin p; aceste segmente vor deveni consecutive în ordinea totală, când s este şters. Dacă aceste segmente nu se intersectează, linia 11 şterge segmentul s

din ordinea totală. În final, dacă nu este găsită nici o intersecţie prin prelucrarea celor $2n$ puncte eveniment, atunci linia 12 returnează FALS.

Corectitudinea

Următoarea teoremă arată că INTERSECŢIA-ORICĂROR-SEGMENTE este corect.

Teorema 35.1 Apelul funcţiei INTERSECŢIA-ORICĂROR-SEGMENTE(S) returnează ADEVĂRAT dacă, şi numai dacă, există o intersecţie printre segmentele lui S.

Demonstraţie. Procedura poate fi incorectă dacă, şi numai dacă, returnează ADEVĂRAT când nu există intersecţie sau dacă returnează FALS când există, cel puţin o intersecţie. Primul caz nu poate să apară deoarece INTERSECŢIA-ORICĂROR-SEGMENTE returnează ADEVĂRAT numai dacă găseşte o intersecţie între două segmente de intrare.

Pentru a arăta că ultimul caz nu poate să apară, să presupunem că există, cel puţin, o intersecţie şi algoritmul INTERSECŢIA-ORICĂROR-SEGMENTE returnează FALS. Fie p cel mai din stânga punct de intersecţie, eliminând egalităţile prin alegerea unui punct cu cea mai mică coordonată-y, şi fie a şi b segmentele care se intersectează în p. Întrucât nu apare nici o intersecţie în stânga lui p, ordinea dată de T este corectă pentru toate punctele din stânga lui p. Există un capăt de segment q pe dreapta de baleiere z care este punctul eveniment la care a şi b devin consecutive în ordinea totală. Dacă p este pe dreapta de baleiere z atunci $q = p$. Dacă p nu este pe dreapta de baleiere z atunci q este în stânga faţă de p. În ambele cazuri, ordinea dată de T este corectă înainte de procesarea lui q. (Aici suntem siguri că p este cel mai de jos dintre cele mai din stânga puncte de intersecţie. Datorită ordinii în care sunt procesate punctele eveniment, chiar dacă p este pe dreapta de baleiere z şi există un alt punct de intersecţie p' pe z, punctul eveniment $q = p$, procesat înaintea celeilalte intersecţii p' poate interfera cu ordinea totală T.) Există doar două posibilităţi de a acţiona în punctul eveniment q:

1. Unul dintre a şi b este inserat în T şi celălalt segment este deasupra sau dedesubtul acestuia în ordinea totală. Liniile 4–7 detectează acest caz.

2. Segmentele a şi b sunt deja în T şi un segment aflat între ele în ordinea totală a fost şters, astfel a şi b devenind consecutive. Liniile 8–11 detectează acest caz.

În ambele cazuri, este găsită intersecţia p, contrazicând presupunerea că procedura returnează FALS. ■

Timpul de execuţie

Dacă există n segmente în mulţimea S, atunci INTERSECŢIA-ORICĂROR-SEGMENTE se execută într-un timp $O(n \lg n)$. Linia 1 necesită un timp $O(1)$. Linia 2 necesită un timp $O(n \lg n)$, folosind sortarea prin interclasare sau heapsort. Deoarece există $2n$ puncte eveniment, ciclul **pentru** din liniile 3–11 se execută de cel mult $2n$ ori. Fiecare iteraţie necesită un timp $O(\lg n)$ deoarece fiecare operaţie arbore-roşu-negru necesită un timp $O(\lg n)$ şi, folosind metoda din secţiunea 35.1, fiecare test de intersecţie necesită un timp $O(1)$. Astfel, timpul total este $O(n \lg n)$.

Exerciţii

35.2-1 Arătaţi că pot exista $\Theta(n^2)$ intersecţii într-o mulţime de n segmente de dreaptă.

35.2-2 Fiind date două segmente a şi b care nu se intersectează şi care sunt comparabile în x, arătaţi cum se folosesc produsele încrucişate pentru a determina, într-un timp $O(1)$, care dintre relaţiile $a >_x b$ sau $b >_x a$ are loc.

35.2-3 Profesorul Maginot ne sugerează să modificăm INTERSECŢIA-ORICĂROR-SEGMENTE, astfel încât, înloc să se termine la găsirea unei intersecţii, să afişeze segmentele care se intersectează şi să continue cu următoarea iteraţie a ciclului **pentru**. Profesorul numeşte procedura care se obţine AFIŞEAZĂ-INTERSECŢIA-SEGMENTELOR şi afirmă că ea afişează toate intersecţiile, de la stânga la dreapta, aşa cum apar ele în mulţimea segmentelor. Arătaţi că profesorul greşeşte în două privinţe dând o mulţime de segmente pentru care prima intersecţie găsită de AFIŞEAZĂ-INTERSECŢIA-SEGMENTELOR nu este cea mai din stânga şi o mulţime de segmente pentru care AFIŞEAZĂ-INTERSECŢIA-SEGMENTELOR eşuează în determinarea tuturor intersecţiilor.

35.2-4 Daţi un algoritm de timp $O(n \lg n)$ care determină dacă un poligon cu n vârfuri este simplu.

35.2-5 Daţi un algoritm de timp $O(n \lg n)$ care determină dacă două poligoane simple, cu un număr total de n vârfuri, se intersectează.

35.2-6 Un ***disc*** este alcătuit dintr-un cerc plus interiorul acestuia şi este reprezentat prin centrul şi raza sa. Două discuri se intersectează dacă au un punct comun. Daţi un algoritm de timp $O(n \lg n)$ care determină dacă oricare două discuri, dintr-o mulţime de n discuri, se intersectează.

35.2-7 Fiind dată o mulţime de n segmente de dreaptă având în total k intersecţii, arătaţi cum pot fi determinate toate cele k intersecţii într-un timp $O((n + k) \lg n)$.

35.3. Determinarea învelitorii convexe

Învelitoarea convexă a unei mulţimi de puncte Q este poligonul convex de arie minimă P, pentru care fiecare punct din Q este sau pe frontiera lui P, sau în interiorul acestuia. Notăm învelitoarea convexă a lui Q prin $\hat{\text{IC}}(Q)$. Intuitiv, ne putem gândi la fiecare punct din Q ca fiind un cui fixat pe o scândură. Atunci, învelitoarea convexă este curba formată de o bandă de cauciuc strânsă care înconjoară toate cuiele. În figura 35.6, se prezintă o mulţime de puncte şi învelitoarea lor convexă.

În această secţiune vom prezenta doi algoritmi care determină învelitoarea convexă a unei mulţimi de n puncte. Ambii algoritmi au ca ieşire vârfurile învelitorii convexe ordonate în sens trigonometric. Primul, cunoscut sub numele de scanarea lui Graham, se execută într-un timp $O(n \lg n)$. Al doilea, numit potrivirea lui Jarvis, se execută într-un timp $O(nh)$, unde h este numărul de vârfuri ale învelitorii convexe. După cum s-a putut observa în figura 35.6, fiecare vârf al lui $\hat{\text{IC}}(Q)$ este un punct din Q. Ambii algoritmi exploatează această proprietate, hotărând care vârfuri din Q sunt păstrate ca şi vârfuri ale învelitorii convexe şi care vârfuri din Q sunt eliminate.

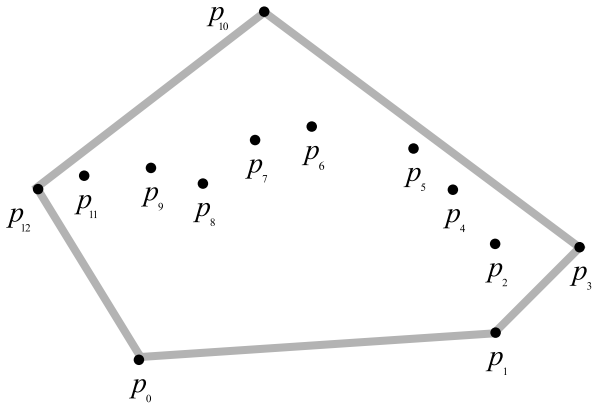


Figura 35.6 O mulţime de puncte Q şi învelitoarea lor convexă $\hat{\text{IC}}(Q)$ desenată cu gri.

De fapt, există câteva metode care calculează învelitori convexe în timpi $O(n \lg n)$. Atât scanarea lui Graham cât şi potrivirea lui Jarvis folosesc o tehnică numită "baleiere rotaţională", procesând vârfurile în ordinea unghiurilor polare pe care le formează cu vârful de referinţă. Alte metode includ următorele:

- În ***metoda incrementală***, punctele sunt ordonate de la stânga la dreapta, producând secvenţa $\langle p_1, p_2, ..., p_n \rangle$. La etapa i, învelitoarea convexă $\hat{\text{IC}}(\{p_1, p_2, ..., p_{i-1}\})$ a celor mai din stânga $i-1$ puncte este actualizată corespunzător celui de-al i-lea punct de la stânga, formând astfel $\hat{\text{IC}}(\{p_1, p_2, ..., p_i\})$. Aşa cum se cere să arătaţi în exerciţiul 35.3-6, această metodă poate fi implementată astfel încât timpul total de execuţie să fie $O(n \lg n)$.

- În ***metoda divide şi stăpâneşte***, în timpul $\Theta(n)$ mulţimea de n puncte este împărţită în două submulţimi, una cu cele mai din stânga $\lceil n/2 \rceil$ puncte şi una cu cele mai din dreapta $\lfloor n/2 \rfloor$ puncte. Învelitorile convexe ale submulţimilor sunt calculate recursiv, şi apoi este folosită o metodă inteligentă de a combina învelitorile într-un timp $O(n)$.

- ***Metoda trunchiază-şi-caută*** este similară cu cel mai defavorabil caz al algoritmului median de timp liniar din secţiunea 10.3. Aceasta găseşte porţiunea superioară (sau "lanţul superior") al învelitorii convexe prin eliminarea repetată a unei părţi constante dintre punctele rămase până când nu mai rămâne decât lanţul superior al învelitorii convexe. Lanţul inferior este determinat în mod analog. Această metodă este, asimptotic, cea mai rapidă: dacă învelitoarea convexă conţine h vârfuri, execuţia necesită doar un timp $O(n \lg h)$.

Determinarea învelitorii convexe a unei mulţimi de puncte este, în sine, o problemă interesantă. Mai mult, algoritmi pentru multe probleme de geometrie computaţională încep prin determinarea unei învelitori convexe. Să analizăm, de exemplu, în spaţiul bidimensional, ***problema perechii celei mai îndepărtate***: avem dată o mulţime de n puncte din plan şi vrem să găsim două puncte a căror distanţă unul faţă de celălalt este maximă. Aşa cum cere exerciţiul 35.3-3, aceste două puncte trebuie să fie vârfuri ale învelitorii convexe. Deşi nu vrem să demonstrăm aici, afirmăm că perechea celui mai îndepărtat de vârf a unui poligon convex cu n vârfuri poate fi găsită într-un timp $O(n)$. Deci, calculând învelitoarea convexă a n puncte

de intrare într-un timp $O(n \lg n)$ şi apoi găsind perechea cea mai departată dintre vârfurile poligonului convex rezultat, putem găsi într-un timp $O(n \lg n)$ perechea celui mai îndepărtat punct din orice mulţime de n puncte.

Scanarea Graham

Scanarea Graham rezolvă problema învelitorii convexe prin întreţinerea unei stive S de puncte posibile. Fiecare punct din mulţimea de intrare Q este pus o dată pe stivă şi punctele care nu sunt vârfuri ale lui $\hat{\text{I}}\text{C}(Q)$ sunt, eventual, scoase din stivă. La terminarea algoritmului, stiva S conţine exact vârfurile lui $\hat{\text{I}}\text{C}(Q)$, ordonate după apariţia lor pe frontieră în sens trigonometric.

Procedura SCANAREA-GRAHAM primeşte ca intrare o mulţime de puncte Q, unde $|Q| \geq 3$. Ea apelează funcţia VÂRF(S) care returnează punctul din vârful stivei S, fără a modifica stiva, şi funcţia URMĂTORUL-VÂRF(S) care returnează punctul cu o intrare mai jos decât vârful stivei S, fără a modifica stiva. Aşa cum vom demonstra imediat, stiva S returnată de procedura SCANAREA-GRAHAM conţine, de jos în sus, în ordine trigonometrică, exact vârfurile lui $\hat{\text{I}}\text{C}(Q)$.

SCANAREA-GRAHAM(Q)
1: fie p_0 punctul din Q cu coordonata y minimă sau cel mai din stânga astfel de punct, în caz că există mai multe
2: fie $\langle p_1, p_2, ..., p_m \rangle$ punctele rămase din Q, ordonate după unghiurile polare în jurul lui p_0, în sens trigonometric (dacă mai multe puncte au acelaşi unghi, atunci se păstrează punctul cel mai îndepărtat faţă de p_0)
3: *vârf*$[S] \leftarrow 0$
4: PUNE-ÎN-STIVĂ(p_0, S)
5: PUNE-ÎN-STIVĂ(p_1, S)
6: PUNE-ÎN-STIVĂ(p_2, S)
7: **pentru** $i \leftarrow 3$ **la** m **execută**
8: **cât timp** unghiul format de punctele URMĂTORUL-VÂRF(S), VÂRF(S) şi p_i face o întoarcere, dar nu spre stânga **execută**
9: SCOATE-DIN-STIVĂ(S)
10: PUNE-ÎN-STIVĂ(p_i, S)
11: **returnează** S

În figura 35.7 se ilustrează evoluţia procedurii SCANAREA-GRAHAM. Linia 1 alege punctul p_0 ca punct având coordonata y cea mai mică, în caz de egalitate alegând punctul cel mai din stânga. Deoarece nu există nici un punct în Q care este mai jos decât p_0 şi orice alt punct cu aceeaşi coordonată y este la dreapta lui p_0, acesta este un vârf al lui $\hat{\text{I}}\text{C}(Q)$. Linia 2 sortează punctele rămase din Q după unghiurile polare relative la p_0, folosind aceeaşi metodă – compararea produselor încrucişate – ca în exerciţiul 35.1-2. Dacă două sau mai multe puncte au acelaşi unghi polar relativ la p_0, atunci toate aceste puncte, cu excepţia celui mai îndepărtat, sunt combinaţii convexe ale lui p_0 cu punctul cel mai îndepărtat şi, astfel, le eliminăm pe toate din discuţie. Notăm cu m numărul punctelor diferite de p_0 care rămân. Unghiul polar, măsurat în radiani, al fiecărui punct din Q relativ la p_0 este în intervalul $[0, \pi/2)$. Deoarece unghiurile polare cresc în sens trigonometric, punctele sunt sortate relativ la p_0 în sens trigonometric. Notăm această secvenţă ordonată de puncte prin $\langle p_1, p_2, ..., p_m \rangle$. Observaţi că punctele p_1 şi p_m sunt vârfuri ale lui $\hat{\text{I}}\text{C}(Q)$ (vezi exerciţiul 35.3-1). În figura 35.7(a) sunt prezentate punctele figurii 35.6, $\langle p_1, p_2, ..., p_{12} \rangle$ cu unghiurile polare ordonate relativ la p_0.

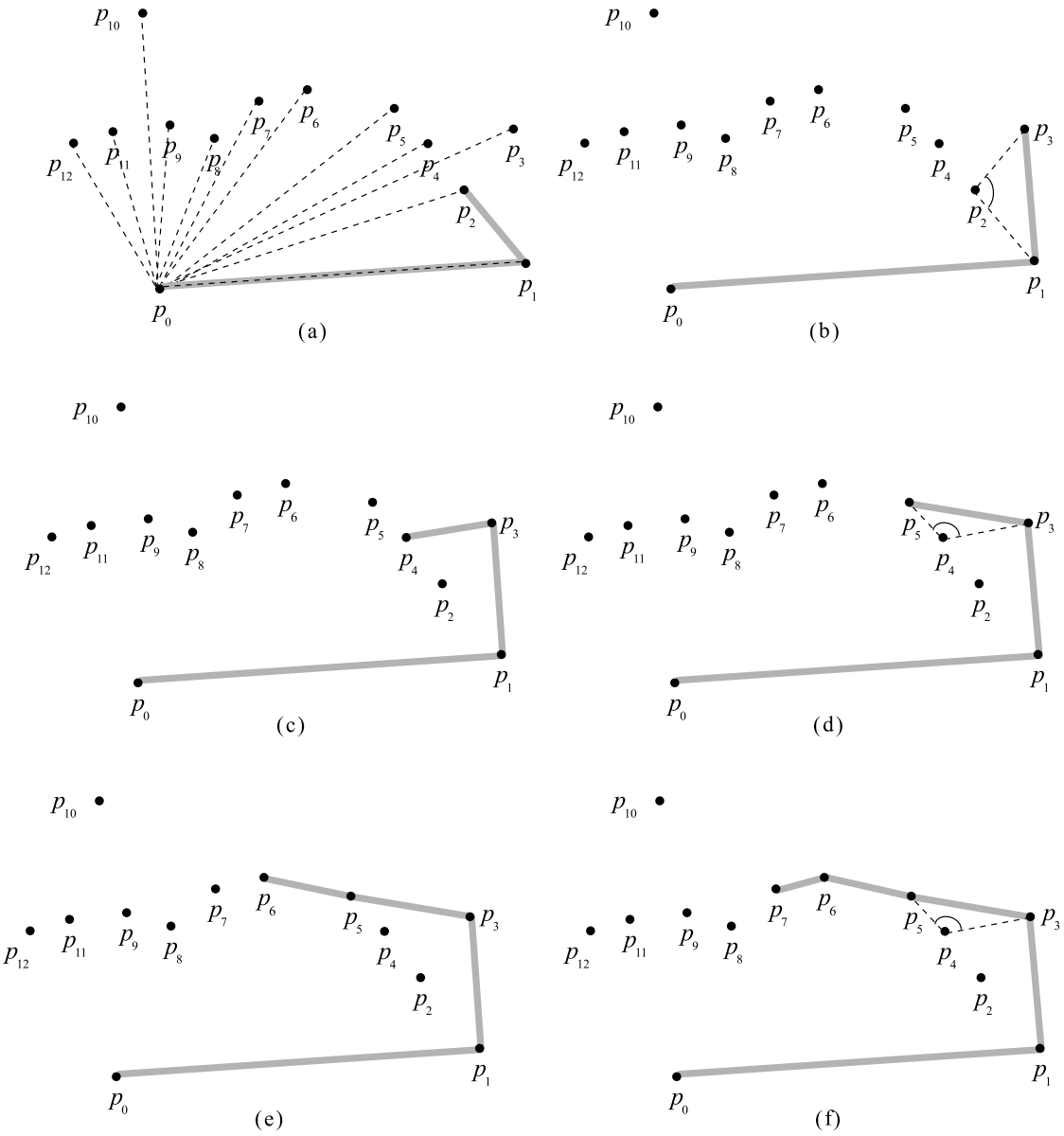
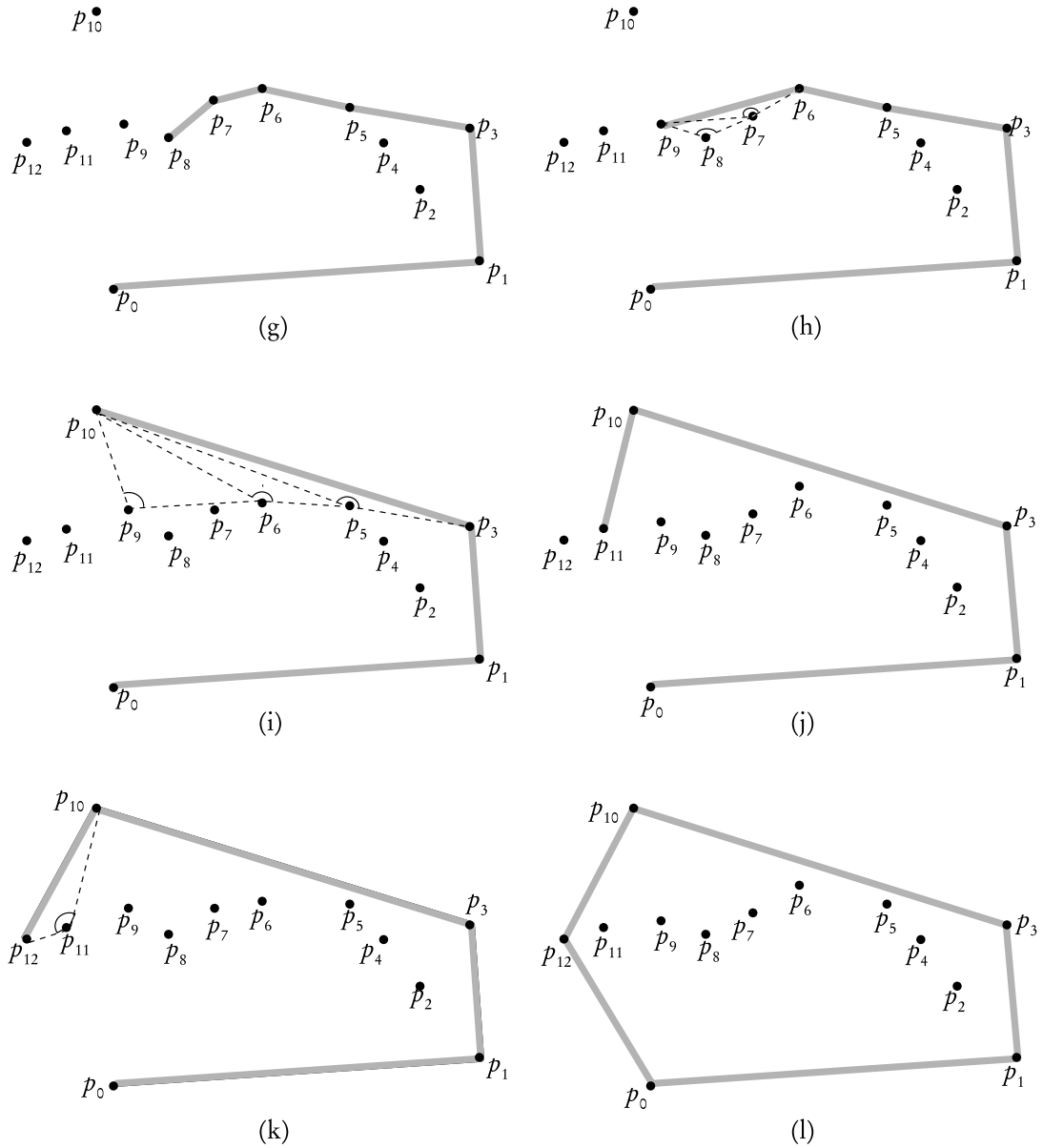


Figura 35.7 Execuţia algoritmului SCANAREA-GRAHAM pentru mulţimea Q din figura 35.6. Învelitoarea convexă curentă din stiva S este reprezentată la fiecare pas cu gri. **(a)** Unghiurile polare din $\langle p_1, p_2, ..., p_{12} \rangle$ ordonate relativ la p_0 şi stiva iniţială S conţinând p_0, p_1 şi p_2. **(b)-(k)** Stiva S după fiecare iteraţie a ciclului **pentru** din liniile 7–10. Liniile punctate arată întoarcerile care nu sunt spre stânga, cauzând scoaterea punctelor din stivă. De exemplu, în (h), întoarcerea spre dreapta din unghiul $\angle p_7 p_8 p_9$ conduce la scoaterea lui p_8 din stivă, şi apoi întoarcerea spre dreapta din unghiul $\angle p_6 p_7 p_9$ conduce la scoaterea lui p_7. **(l)** Învelitoarea convexă returnată de procedură, aceeaşi ca în figura 35.6.



Restul procedurii foloseşte stiva S. Liniile 3–6 iniţializează stiva, de jos în sus, cu primele trei puncte p_0, p_1 şi p_2. În figura 35.7(a) se prezintă stiva iniţială S. Ciclul **pentru** din liniile 7–10 se execută o dată pentru fiecare punct al secvenţei $\langle p_3, p_4, ..., p_m \rangle$. Intenţia este ca, după procesarea punctului p_i, stiva S să conţină, de jos în sus, vârfurile lui $\text{ÎC}(\{p_0, p_1, ..., p_i\})$ ordonate în sens trigonometric. Ciclul **cât timp**, din liniile 8–9, scoate din stivă punctele care nu fac parte din învelitoarea convexă. Când parcurgem învelitoarea convexă în sens trigonometric trebuie să facem întoarceri la stânga în fiecare vârf. Deci, de fiecare dată când ciclul **cât timp** găseşte un vârf în care nu facem o întoarcere la stânga, vârful este scos din stivă. (Verificând că o întoarcere

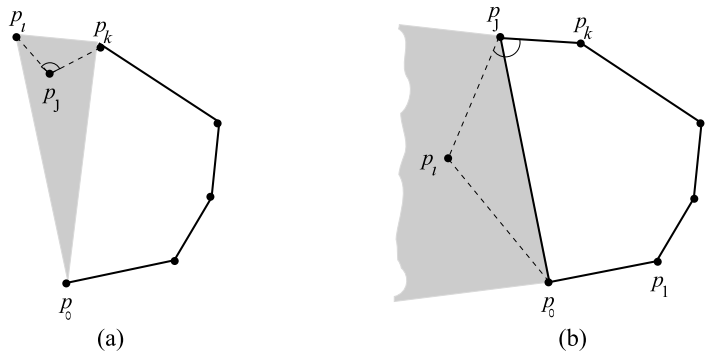


Figura 35.8 Două situaţii esenţiale în demonstrarea corectitudinii algoritmului SCANAREA-GRAHAM. **(a)** Se arată că, în SCANAREA-GRAHAM, un punct scos din stivă nu este un vârf din $\hat{\text{IC}}(Q)$. Dacă vârful p_j este scos din stivă deoarece unghiul $\angle p_k p_j p_i$ nu face o întoarcere la stânga, atunci triunghiul haşurat, $\triangle p_0 p_k p_i$, conţine punctul p_j. Deci punctul p_j nu este un vârf din $\hat{\text{IC}}(Q)$. **(b)** Dacă punctul p_i este pus în stivă, atunci în unghiul $\angle p_k p_j p_i$ trebuie să fie o întoarcere la stânga. Punctul p_i trebuie să fie în regiunea haşurată deoarece p_i urmează după p_j în ordinea unghiurilor polare ale punctelor şi datorită modului în care a fost ales p_0. Dacă punctele din stivă formează un poligon convex înainte de operaţia de punere, atunci ele trebuie să formeze un poligon convex şi după aceea.

nu este spre stânga şi nu doar că este spre dreapta, se exclude posibilitatea ca un unghi alungit să facă parte din învelitoarea convexă rezultată. Aceasta este, tocmai, ceea ce dorim deoarece fiecare vârf al unui poligon convex nu trebuie să fie o combinaţie de alte vâfuri ale poligonului.) După ce scoatem din stivă toate vârfurile care nu au întoarceri spre stânga, când mergem către punctul p_i, punem p_i în stivă. În figura 35.7(b)-(k), se prezintă starea stivei S după fiecare iteraţie a ciclului **pentru**. În final, în linia 11, SCANAREA-GRAHAM returnează stiva S. În figura 35.7(l) se prezintă învelitoarea convexă corespunzătoare.

Următoarea teoremă demonstrează formal corectitudinea procedurii SCANAREA-GRAHAM.

Teorema 35.2 (Corectitudinea algoritmului scanarea Graham) Dacă procedura SCANAREA-GRAHAM este executată pentru o mulţime de puncte Q, $|Q| \geq 3$, atunci un punct din Q este pe stiva S la terminare dacă, şi numai dacă, este vârf al învelitorii convexe $\hat{\text{IC}}(Q)$.

Demonstraţie. Aşa cum am observat mai sus, un vârf care este o combinaţie convexă a lui p_0 şi alte vârfuri din Q, nu este vârf al lui $\hat{\text{IC}}(Q)$. Un astfel de vârf nu este inclus în secvenţa $\langle p_1, p_2, ..., p_m \rangle$ şi deci el nu poate să apară în stiva S.

Dificultatea acestei demonstraţii constă în cele două situaţii ilustrate în figura 35.8. Cadrul (a) tratează întoarcerile care nu sunt la stânga, iar cadrul (b) tratează întoarcerile la stânga.

Arătăm, mai întâi, că fiecare punct scos din stiva S nu este un vârf al învelitorii $\hat{\text{IC}}(Q)$. Presupunem că punctul p_j este scos din stivă deoarece unghiul $\angle p_k p_j p_i$ nu face o întoarcere la stânga, conform figurii 35.8(a). Deoarece parcurgem punctele în ordinea crescătoare a unghiurilor polare relative la punctul p_0, există un triunghi $\triangle p_0 p_i p_k$ cu punctul p_j sau în interiorul lui sau pe segmentul $\overline{p_i p_k}$. În ambele cazuri, punctul p_j nu poate fi un punct din $\hat{\text{IC}}(Q)$.

Arătăm acum că, în final, fiecare punct din stiva S este un vârf din $\hat{\text{IC}}(Q)$. Începem prin a demonstra următoarea afirmaţie: SCANAREA-GRAHAM susţine invariantul că punctele din stiva S fac parte, întotdeauna, din vârfurile unui poligon convex, ordonate în sens trigonometric.

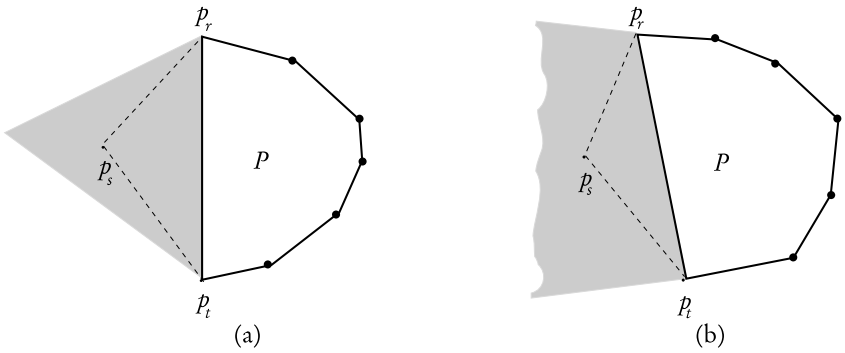


Figura 35.9 Adăugând un punct unui poligon convex P, în regiunea haşurată, se obţine un alt poligon convex. Regiunea haşurată este mărginită de o latură $\overline{p_r p_t}$ şi prelungirile a două laturi adiacente. **(a)** Regiunea haşurată este mărginită. **(b)** Regiunea haşurată este nemărginită.

Imediat după execuţia liniei 6, afirmaţia este valabilă deoarece punctele p_0, p_1 şi p_2 formează un poligon convex. Urmărim, acum, cum se modifică stiva S în timpul execuţiei algoritmului SCANAREA-GRAHAM. Punctele sunt sau scoase, sau puse în stivă. În primul caz, ne bazăm pe o simplă proprietate geometrică: dacă un vârf este eliminat dintr-un poligon convex, poligonul rezultat este convex. Deci scoaterea unui punct din stiva S păstrează invariantul.

Înainte de a trata cazul în care un punct este pus în stivă, să examinăm o altă proprietate geometrică, ilustrată în figura 35.9(a) şi (b). Fie P un poligon convex, alegem o latură oarecare $\overline{p_r p_t}$ din P. Să analizăm regiunea mărginită de $\overline{p_r p_t}$ şi prelungirile celor două laturi adiacente. (În funcţie de unghiurile relative ale laturilor adiacente, regiunea poate fi mărginită, ca în regiunea haşurată din cadrul (a), sau nemărginită, ca în cadrul (b).) Dacă adăugăm orice punct p_s din această regiune la P ca un nou vârf, înlocuind latura $\overline{p_r p_t}$ cu laturile $\overline{p_r p_s}$ şi $\overline{p_s p_t}$, poligonul rezultat este convex.

Considerăm acum că punctul p_i este pus pe stiva S. Referindu-ne la figura 35.8(b), fie p_j punctul din vârful stivei S, chiar înainte de a fi pus p_i, şi fie p_k predecesorul lui p_j în S. Afirmăm că p_i trebuie să fie în zona haşurată din figura 35.8(b), care corespunde direct cu regiunea haşurată din figura 35.9(b). Punctul p_i trebuie să fie în partea haşurată faţă de prelungirea lui $\overline{p_k p_j}$ deoarece unghiul $\angle p_k p_j p_i$ face o întoarcere la stânga. El trebuie să fie în partea haşurată a lui $\overline{p_0 p_j}$ deoarece, în ordinea unghiurilor polare, p_i este după p_j. Mai mult, după cum am ales punctul p_0, punctul p_i trebuie să fie în partea haşurată faţă de prelungirea lui $\overline{p_0 p_1}$. Deci p_i este în regiunea haşurată, şi, după punerea lui p_i în stiva S, punctele din S formează un poligon convex. Acum, demonstraţia afirmaţiei este completă.

Deci, în finalul procedurii SCANAREA-GRAHAM punctele din Q care sunt în stiva S formează vârfurile unui poligon convex. Am arătat că toate punctele care nu sunt în S nu sunt din $\hat{\text{IC}}(Q)$ sau, în mod echivalent, că toate punctele care sunt din $\hat{\text{IC}}(Q)$ sunt în S. Întrucât S conţine doar vârfuri din Q şi punctele sale formează un poligon convex, ele trebuie să formeze $\hat{\text{IC}}(Q)$. ■

Arătăm acum că timpul de execuţie al algoritmului SCANAREA-GRAHAM este $O(n \lg n)$, unde $n = |Q|$. Linia 1 necesită un timp $\Theta(n)$. Linia 2 necesită un timp $O(n \lg n)$ folosind sortarea prin interclasare sau algoritmul heapsort pentru ordonarea unghiurilor polare şi metoda produsului încrucişat, din secţiunea 35.1, pentru compararea unghiurilor. (Ştergerea tuturor punctelor cu

acelaşi unghi polar, cu excepţia celui mai îndepărtat punct, poate fi realizată într-un timp $O(n)$). Liniile 3–6 necesită un timp $O(1)$. Ciclul **pentru** este executat de cel mult $n-3$ ori deoarece $m \leq n-1$. Întrucât PUNE-ÎN-STIVĂ necesită un timp $O(1)$, fiecare iteraţie necesită exclusiv un timp $O(1)$ din timpul consumat de ciclul **cât timp**, din liniile 8–9. Deci, în total, ciclul **pentru** necesită exclusiv un timp $O(n)$ pentru ciclul **cât timp** pe care îl conţine.

Am folosit metoda agregată din analiza amortizată pentru a arăta că ciclul **cât timp** necesită un timp de execuţie total de $O(n)$. Pentru $i = 0, 1, ..., m$, fiecare punct p_i este pus exact o dată în stiva S. La fel ca şi în analiza procedurii SCOATERE-MULTIPLĂ-DIN-STIVĂ din secţiunea 18.1, observăm că există, cel mult, o operaţie SCOATE-DIN-STIVĂ pentru fiecare operaţie PUNE-ÎN-STIVĂ. Exact trei puncte – p_0, p_1 şi p_m – nu sunt scoase din stivă niciodată, deci se efectuează cel mult $m-2$ operaţii SCOATE-DIN-STIVĂ în total. Fiecare iteraţie a ciclului **cât timp** execută o operaţie SCOATE-DIN-STIVĂ şi deci există, în total, cel mult $m-2$ iteraţii ale ciclului **cât timp**. Întrucât testul din linia 8 necesită un timp $O(1)$, fiecare apel SCOATE-DIN-STIVĂ necesită un timp $O(1)$ şi $m \leq n-1$, timpul total necesar execuţiei ciclului **cât timp** este $O(n \lg n)$.

Potrivirea lui Jarvis

Potrivirea lui Jarvis determină învelitoarea convexă a unei mulţimi de puncte Q folosind o tehnică cunoscută sub numele de ***împachetarea pachetului*** (sau ***împachetarea cadoului***). Timpul de execuţie al algoritmului este $O(nh)$, unde h este numărul de vârfuri din $\hat{\text{I}}\text{C}(Q)$. Când h este $O(\lg n)$, atunci potrivirea lui Jarvis este, asimptotic, mai rapidă decât scanarea lui Graham.

Intuitiv, potrivirea lui Jarvis simulează împachetarea unei bucăţi de hârtie bine întinsă în jurul mulţimii Q. Începem prin a lega capătul hârtiei la cel mai de jos punct al mulţimii, adică acelaşi punct p_0 cu care am început scanarea Graham. Acest punct este un vârf al învelitorii convexe. Împingem hârtia spre dreapta pentru a o întinde bine, iar apoi o împingem în sus până când atinge un alt punct. Acest vârf trebuie să fie, de asemenea, pe învelitoarea convexă. Păstrând hârtia întinsă, continuăm în acest fel, înconjurând mulţimea de vârfuri până când ajungem înapoi la punctul de pornire p_0.

Formal, potrivirea lui Jarvis construieşte o secvenţă $H = \langle p_0, p_1, ..., p_{h-1} \rangle$ din punctele din $\hat{\text{I}}\text{C}(Q)$. Începem cu punctul p_0. După cum se vede în figura 35.10, următorul vârf al învelitorii convexe, p_1, are cel mai mic unghi polar în raport cu p_0. (În caz de egalitate, vom alege punctul cel mai îndepărtat de p_0.) În mod analog, p_2 este cel mai mic unghi polar în raport cu p_1 şi aşa mai departe. Când ajungem la vârful cel mai de sus, fie acesta p_k (eliminând egalităţile prin alegerea celui mai îndepărtat vârf), am construit, aşa cum se poate vedea şi în figura 35.10, ***lanţul drept*** al învelitorii $\hat{\text{I}}\text{C}(Q)$. Pentru construirea ***lanţului stâng***, pornim de la p_k şi alegem punctul p_{k+1}, punctul cu cel mai mic unghi polar în raport cu p_k, dar de pe ***axa-x negativă***. Continuăm să formăm lanţul stâng prin luarea unghiurilor polare de pe axa-x negativă până când revenim în vârful originar p_0.

Putem implementa potrivirea lui Jarvis ca o deplasare conceptuală în jurul învelitorii convexe, adică fără să construim, separat, lanţurile drept şi stâng. Astfel de implementări, de obicei, păstrează ruta unghiului ultimei laturi alese pe învelitoarea convexă şi necesită ca secvenţa de unghiuri ale laturilor învelitorii să fie în ordine strict crescătoare (în intervalul 0–2π radiani). Avantajul construirii separate a lanţurilor este că nu avem nevoie de calculul explicit al unghiurilor; sunt suficiente tehnicile din secţiunea 35.1 pentru compararea unghiurilor.

Timpul de execuţie pentru potrivirea lui Jarvis, la o implementare corectă, este $O(nh)$. Pentru fiecare dintre cele h vârfuri din $\hat{\text{I}}\text{C}(Q)$, găsim vârful cu unghiul polar minim. Fiecare comparaţie

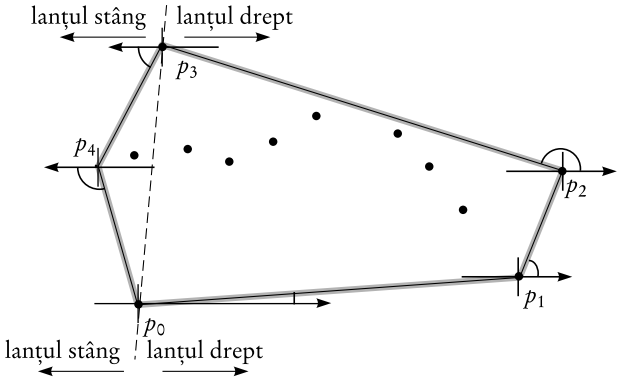


Figura 35.10 Funcţionarea algoritmului de potrivire a lui Jarvis. Primul vârf ales este cel mai de jos punct p_0. Următorul vârf, p_1, are unghiul polar cel mai mic dintre toate punctele, în raport cu p_0. Apoi, p_2 are cel mai mic unghi polar în raport cu p_1. Lanţul drept merge la fel de sus ca şi cel mai de sus punct p_3. Atunci, lanţul drept este construit prin căutarea celor mai mici unghiuri polare în raport cu axa-x negativă.

între unghiuri polare, folosind tehnicile din secţiunea 35.1, necesită un timp $O(1)$. Aşa cum este arătat în secţiunea 10.1, putem determina minimul a n valori într-un timp $O(n)$ dacă fiecare comparaţie necesită un timp $O(1)$. Deci potrivirea lui Jarvis necesită un timp $O(nh)$.

Exerciţii

35.3-1 Demonstraţi că, în procedura SCANAREA-GRAHAM, punctele p_1 şi p_m trebuie să fie vârfuri din $\hat{\text{I}}\text{C}(Q)$.

35.3-2 Considerăm un model de calcul care permite adunarea, compararea şi înmulţirea, şi care conţine marginea inferioară $\Omega(n \lg n)$ pentru ordonarea a n numere. Demonstraţi că $\Omega(n \lg n)$ este o margine inferioară pentru calcularea, în ordine, a vârfurilor învelitorii convexe a unei mulţimi de n puncte într-un astfel de model.

35.3-3 Fiind dată o mulţime de puncte Q, demonstraţi că perechea punctelor celor mai îndepărtate unul faţă de celălalt trebuie să fie vârfuri din $\hat{\text{I}}\text{C}(Q)$.

35.3-4 Fie un poligon P şi un punct q pe frontiera acestuia. Numim ***umbra*** lui q mulţimea de puncte r al căror segment $\overline{qr}$ se află, în întregime, pe frontiera sau în interiorul lui P. Un poligon P este ***în formă de stea*** dacă există în interiorul său un punct p care se află în umbra oricărui punct de pe frontiera lui P. Mulţimea tuturor acestor puncte p se numeşte ***nucleul*** lui P. (Vezi figura 35.11.) Fiind dat un poligon P cu n vârfuri, în formă de stea, specificat prin vârfurile lui ordonate în sens trigonometric, arătaţi cum se calculează $\hat{\text{I}}\text{C}(P)$ într-un timp $O(n)$.

35.3-5 În ***problema învelitorii convexe în timp real***, avem dată o mulţime Q de n puncte pe care le primim pe rând. La primirea fiecărui punct, calculăm învelitoarea convexă a punctelor cunoscute până la momentul curent. Evident, se poate executa pentru fiecare punct scanarea Graham, fiind necesar un timp de execuţie de $O(n^2 \lg n)$. Arătaţi cum se rezolvă problema învelitorii convexe în timp real, astfel încât timpul de execuţie să fie $O(n^2)$.

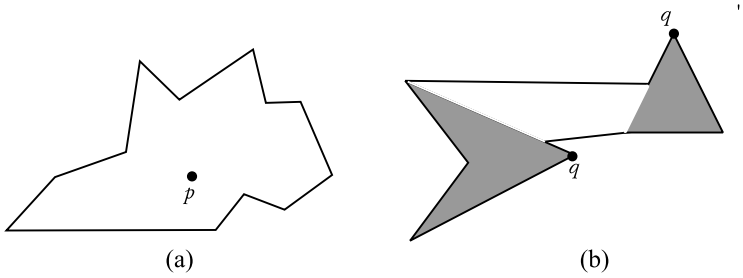


Figura 35.11 Definiţia unui poligon în formă de stea, pentru exerciţiul 35.3-4. **(a)** Un poligon în formă de stea. Segmentul din punctul p spre orice punct q de pe frontiera poligonului intersectează frontiera numai în q. **(b)** Un ploligon care nu este în formă de stea. Regiunea haşurată din stânga este umbra lui q iar regiunea umbrită din dreapta este umbra lui q'. Nucleul este mulţimea vidă, deoarece aceste regiuni sunt disjuncte.

35.3-6 $\star$ Arătaţi cum se implementează metoda incrementală pentru calcularea învelitorii convexe a n puncte, astfel încât timpul de execuţie să fie $O(n \lg n)$.

35.4. Determinarea celei mai apropiate perechi de puncte

Analizăm problema determinării celei mai apropiate perechi de puncte într-o mulţime Q de $n \geq 2$ puncte. "Cele mai apropiate" se referă la distanţa euclidiană: distanţa dintre punctele $p_1 = (x_1, y_1)$ şi $p_2 = (x_2, y_2)$ este $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Două puncte din mulţimea Q pot coincide, caz în care distanţa dintre ele este zero. Această problemă are aplicaţii, de exemplu, în sistemele de control ale traficului. Un sistem pentru controlul traficului aerian sau maritim are nevoie să ştie care sunt cele mai apropiate două vehicule pentru a detecta o posibilă situaţie de coliziune.

Un algoritm de forţă brută pentru determinarea celei mai apropiate perechi se uită, pur şi simplu, la toate cele $\binom{n}{2} = \Theta(n^2)$ perechi de puncte. În acest capitol vom descrie un algoritm divide şi stăpâneşte pentru această problemă, al cărui timp de execuţie este descris de bine cunoscuta relaţie de recurenţă $T(n) = 2T(n/2) + O(n)$. Deci, acest algoritm foloseşte doar un timp $O(n \lg n)$.

Algoritmul divide şi stăpâneşte

Fiecare apel recursiv al algoritmului necesită, ca intrare, o submulţime $P \subseteq Q$ şi două şiruri X şi Y, fiecare conţinând toate punctele submulţimii de intrare P. Punctele din şirul X sunt ordonate astfel încât coordonatele lor x să fie monoton crescătoare. În mod analog, şirul Y este ordonat monoton crescător după coordonata y. Reţineţi că, pentru a ajunge la timpul limită $O(n \lg n)$, nu ne permitem să facem sortări la fiecare apel recursiv; dacă am face, timpul de execuţie pentru o recurenţă ar fi $T(n) = 2T(n/2) + O(n \lg n)$, iar pentru soluţie $T(n) = O(n \lg^2 n)$. Vom vedea mai târziu cum se foloseşte "presortarea" pentru a menţine şirurile sortate fără a le sorta efectiv la fiecare apel recursiv.

Un apel recursiv cu intrările P, X şi Y verifică întâi dacă $|P| \leq 3$. Dacă da, atunci se va folosi metoda forţei brute descrisă mai sus: consideră toate cele $\binom{|P|}{2}$ perechi de puncte şi returnează perechea cea mai apropiată. Dacă $|P| \geq 3$, apelul recursiv va folosi paradigma divide şi stăpâneşte după cum urmează.

1. **Divide:** Determină o dreaptă verticală d care împarte mulţimea de puncte P în două submulţimi P_S şi P_D, astfel încât $|P_S| = \lceil |P|/2 \rceil$, $|P_D| = \lfloor |P|/2 \rfloor$, toate punctele din P_S sunt pe dreapta d sau în stânga ei şi toate punctele din P_D sunt pe sau în dreapta dreptei d. Şirul X este împărţit în şirurile X_S şi X_D care conţin punctele din P_S şi, respectiv, P_D, sortate monoton crescător după coordonata x. În mod analog, şirul Y este împărţit în şirurile Y_S şi Y_D care conţin punctele din P_S şi, respectiv, P_D, sortate monoton crescător după coordonata y.

2. **Stăpâneşte:** Având împărţită mulţimea P în P_S şi P_D, se fac două apeluri recursive, unul pentru a determina cea mai apropiată pereche de puncte din P_S, şi celălalt pentru a determina cea mai apropiată pereche de puncte din P_D. Intrările pentru primul apel sunt mulţimea P_S şi şirurile X_S şi Y_S; iar pentru al doilea apel intrările sunt P_D, X_D şi Y_D. Fie δ_S şi δ_D valorile returnate pentru distanţele celor mai apropiate perechi din P_S şi, respectiv, P_D şi fie $\delta = \min(\delta_S, \delta_D)$.

3. **Combină:** Cea mai apropiată pereche este cea cu distanţa δ, determinată de unul din apelurile recursive, sau este o pereche de puncte cu un punct în P_S şi celălalt în P_D. Algoritmul determină dacă există o astfel de pereche cu distanţa mai mică decât δ. Observaţi că, dacă există o pereche de puncte cu distanţa mai mică decât δ, atunci ambele puncte ale perechii trebuie să fie, faţă de dreapta d, la distanţa maximă δ. Astfel, conform figurii 35.12(a), ambele trebuie să se situeze într-o regiune de lăţime 2δ, centrată în jurul dreptei verticale d. Pentru a găsi o astfel de pereche, dacă există, algoritmul procedează după cum urmează:

   (a) Construieşte un şir Y', care este şirul Y fără punctele ce sunt în afara regiunii de lăţime 2δ. Şirul Y' este sortat după coordonata y, la fel ca şi Y.

   (b) Pentru fiecare punct p din şirul Y', algoritmul încearcă să găsească punctele din Y' care sunt la o distanţă de cel mult δ unităţi faţă de p. Aşa cum vom vedea imediat, este necesar să fie considerate doar 7 puncte din Y', care urmează după p. Algoritmul calculează distanţa de la p la fiecare dintre cele 7 puncte şi reţine distanţa δ' a perechii celei mai apropiate, găsite dintre toate perechile de puncte din Y'.

   (c) Dacă $\delta' < \delta$, atunci regiunea verticală conţine, într-adevăr, o pereche mai apropiată decât cea care a fost găsită prin apelurile recursive. Sunt returnate această pereche şi distanţa sa δ'. Altfel, sunt returnate cea mai apropiată pereche şi distanţa sa δ, găsite prin apelurile recursive.

În descrierea de mai sus au fost omise unele detalii de implementare care sunt necesare pentru a obţine timpul de execuţie $O(n \lg n)$. După demonstrarea corectitudinii algoritmului, vom prezenta cum se implementează algoritmul pentru a obţine timpul limită dorit.

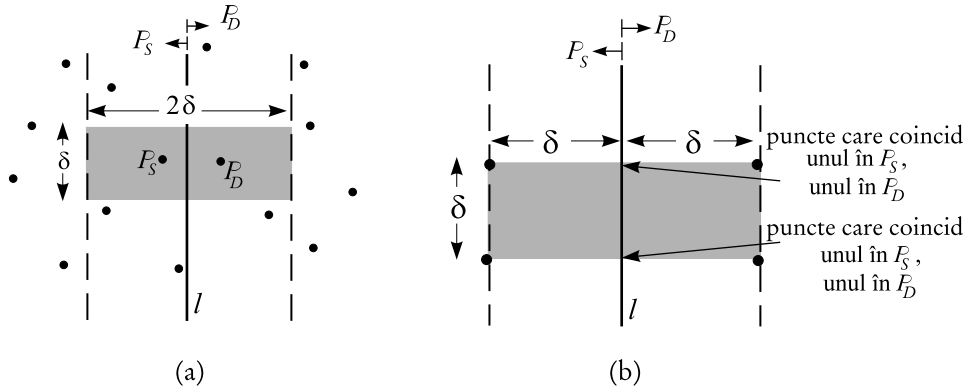


Figura 35.12 Ideea cheie pentru a demonstra că algoritmul pentru determinarea perechii celei mai apropiate necesită verificarea a doar 7 puncte care urmează după fiecare punct din şirul Y'. **(a)** Dacă $p_S \in P_S$ şi $p_D \in P_D$ sunt la mai puţin de δ unităţi distanţă, atunci ele trebuie să se afle în interiorul unui dreptunghi $\delta \times 2\delta$ centrat pe dreapta d. **(b)** Modalitatea de a situa 4 puncte în interiorul unui pătrat $\delta \times \delta$ dacă sunt la mai puţin de δ unităţi distanţă două-câte-două. În stânga sunt 4 puncte din P_S, iar în dreapta sunt 4 puncte din P_D. Acolo pot fi 8 puncte situate în dreptunghiul $\delta \times 2\delta$ dacă punctele de pe dreapta d sunt perechi de puncte care coincid cu un punct în P_S şi unul în P_D.

Corectitudinea

Corectitudinea acestui algoritm pentru determinarea perechii celei mai apropiate este evidentă, cu excepţia a două aspecte. Primul, oprind recursivitatea când $|P| \leq 3$, ne asigurăm că nu vom încerca niciodată să împărţim o mulţime cu un singur punct. Al doilea aspect este că avem nevoie doar de 7 puncte care urmează după fiecare punct p din şirul Y'. Vom demonstra în continuare această proprietate.

Presupunem că la un anumit nivel al recursivităţii, cea mai apropiată pereche de puncte este $p_S \in P_S$ şi $p_D \in P_D$. Astfel, distanţa δ' dintre p_S şi p_D este strict mai mică decât δ. Punctul p_S trebuie să fie pe dreapta d sau în stânga ei şi la o distanţă mai mică de δ unităţi. În mod analog, p_D este pe sau în dreapta dreptei d şi la o distanţă mai mică de δ unităţi. Mai mult, p_S şi p_D se află pe verticală la o distanţă de cel mult δ unităţi unul faţă de celălalt. Deci, aşa cum se arată în figura 35.12(a), p_S şi p_D se află într-un dreptunghi $\delta \times 2\delta$ centrat pe dreapta d. (Desigur, pot fi şi alte puncte în interiorul acestui dreptunghi).

Arătăm în continuare că cel mult 8 puncte din P se pot situa în interiorul acestui dreptunghi $\delta \times 2\delta$. Studiem pătratul $\delta \times \delta$ care reprezintă jumătatea stângă a dreptunghiului. Deoarece toate punctele din P_S sunt la o distanţă de cel puţin δ unităţi, cel mult 4 puncte se pot situa în interiorul acestui pătrat; figura 35.12(b) arată cum. În mod analog, cel mult 4 puncte din P_D se pot situa în interiorul pătratului $\delta \times \delta$ ce reprezintă jumătatea dreaptă a dreptunghiului. Deci cel mult 8 puncte din P se pot situa în interiorul dreptunghiului $\delta \times 2\delta$. (Observăm că, întrucât punctele de pe d pot fi în oricare dintre mulţimile P_S sau P_D, nu pot fi mai mult de 4 puncte pe d. Această limită este atinsă dacă există două perechi de puncte care coincid, fiecare pereche conţinând un punct din P_S şi un punct din P_D; o pereche se află la intersecţia lui d cu partea de sus a dreptunghiului, iar cealaltă pereche se află la intersecţia lui d cu partea de jos a dreptunghiului.)

Am arătat că numai 8 puncte din P se pot situa în dreptunghi, se vede uşor că este necesar să verificăm cel mult 7 puncte care urmează după fiecare punct în şirul Y'. Păstrând presupunerea că cea mai apropiată pereche este p_S şi p_D, presupunem, fără a reduce generalitatea, că p_S îl precede pe p_D în şirul Y'. Atunci, chiar dacă p_S apare cât de devreme este posibil în şirul Y', şi p_D apare cât mai târziu posibil, p_D este într-una din cele 7 poziţii care urmează după p_S. Deci am demonstrat corectitudinea algoritmului pentru determinarea perechii celei mai apropiate.

Implementarea şi timpul de execuţie

Aşa cum s-a observat, scopul nostru este ca timpul de execuţie să fie recurenţa $T(n) = 2T(n/2) + O(n)$, unde $T(n)$ este, desigur, timpul de execuţie al algoritmului pentru o mulţime de n puncte. Dificultatea principală este de a ne asigura că şirurile X_S şi X_D, Y_S şi Y_D, care sunt folosite în apelurile recursive, sunt sortate după coordonatele proprii şi, de asemenea, că şirul Y' este sortat după coordonata y. (Observăm că, dacă şirul care este recepţionat dintr-un apel recursiv este gata sortat, atunci împărţirea mulţimii P în P_S şi P_D se realizează uşor în timp liniar.)

Observaţia cheie a rezolvării problemei este că, la fiecare apel, dorim să formăm un subşir sortat dintr-un şir sortat. Fie un apel particular având ca date de intrare submulţimea P şi şirul Y, sortate după coordonata y. Având împărţit P în P_S şi P_D, trebuie să formăm şirurile Y_S şi Y_D, care sunt sortate după coordonata y. Mai mult, aceste şiruri trebuie formate într-un timp liniar. Metoda poate fi privită ca inversa procedurii MERGE din sortarea prin interclasare din secţiunea 1.3.1: împărţim un şir sortat în două şiruri sortate. Ideea este dată în pseudocodul următor:

```
1: lungime[Y_S] ← lungime[Y_D] ← 0
2: pentru i ← 1 la lungime[Y] execută
3:    dacă Y[i] ∈ P_S atunci
4:       lungime[Y_S] ← lungime[Y_S] + 1
5:       Y[lungime[Y_S]] ← Y[i]
6:    altfel
7:       lungime[Y_D] ← lungime[Y_D] + 1
8:       Y[lungime[Y_D]] ← Y[i]
```

Examinăm punctele din şirul Y în ordine. Dacă un punct $Y[i]$ este în P_S, îl adăugăm în şirul Y_S la sfârşit; altfel îl adăugăm la sfârşitul şirului Y_D. În mod analog, pseudocodul formează şirurile X_S,X_D şi Y'.

Singura întrebare care mai rămâne este cum sortăm punctele la început. Facem o simplă ***presortare*** a lor; adică le sortăm, o dată, pe toate *înainte* de primul apel recursiv. Aceste şiruri sortate sunt date în primul apel recursiv, de acolo ele sunt reduse prin apeluri succesive cât este necesar. Presortarea adaugă un timp $O(n \lg n)$ la timpul de execuţie, dar acum fiecare etapă a recursivităţii necesită un timp liniar exclusiv pentru apelurile recursive. Deci, dacă avem $T(n)$ timpul de execuţie al fiecărei etape recursive şi $T'(n)$ este timpul de execuţie al algoritmului complet, obţinem $T'(n) = T(n) + O(n \lg n)$ şi

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{dacă } n > 3, \\ O(1) & \text{dacă } n \leq 3. \end{cases}$$

Deci $T(n) = O(n \lg n)$ şi $T'(n) = O(n \lg n)$.

Exerciţii

35.4-1 Profesorul Smothers propune o schemă care permite algoritmului determinării celei mai apropiate perechi să verifice doar 5 puncte care urmează după fiecare punct în şirul Y'. Ideea este să plaseze totdeauna puncte de pe dreapta d în mulţimea P_S. Atunci, nu pot fi perechi de puncte coincidente pe dreapta d, cu un punct în P_S şi unul în P_D. Deci, cel mult, 6 puncte pot fi în dreptunghiul $\delta \times 2\delta$. Ce nu este corect în schema profesorului?

35.4-2 Fără a creşte asimptotic timpul de execuţie al algoritmului, arătaţi cum se garantează că mulţimea de puncte care a trecut de primul apel recursiv nu conţine puncte care coincid. Arătaţi că este suficient să se verifice punctele în 6 (nu 7) poziţii din şir care urmează după fiecare punct din Y'.

35.4-3 Distanţa dintre două puncte poate fi definită şi altfel decât euclidian. În plan, ***distanţa*** L_m dintre punctele p_1 şi p_2 este dată de $((x_1 - x_2)^m + (y_1 - y_2)^m)^{1/m}$. Deci distanţa euclidiană este distanţa L_2. Modificaţi algoritmul celei mai apropiate perechi, astfel încât să folosească distanţa L_1, cunoscută sub numele de ***distanţa Manhattan***.

35.4-4 Fiind date două puncte p_1 şi p_2 în plan, $\max(|x_1 - x_2|, |y_1 - y_2|)$ este distanţa L_∞ dintre ele. Modificaţi algoritmul celei mai apropiate perechi să folosească distanţa L_∞.

Probleme

35-1 Niveluri convexe
Fiind dată o mulţime Q de puncte în plan, definim ***nivelurile convexe*** ale lui Q prin inducţie. Primul nivel convex al lui Q se compune din acele puncte din Q care sunt vârfuri din $\hat{\text{I}}\text{C}(Q)$. Pentru $i > 1$, definim mulţimea Q_i care conţine puncte din Q, mai puţin punctele care au fost în nivelurile convexe $1, 2, ..., i-1$. Atunci, al i-lea nivel convex al lui Q este $\hat{\text{I}}\text{C}(Q_i)$ dacă $Q_i \neq \emptyset$ şi nedefinită altfel.

a. Daţi un algoritm cu timpul de execuţie $O(n^2)$ care determină nivelurile convexe ale unei mulţimi de n puncte.

b. Demonstraţi că timpul $\Omega(n \lg n)$ este necesar pentru a calcula nivelurile convexe ale unei mulţimi de n puncte pe orice model de calcul care necesită un timp $\Omega(n \lg n)$ pentru a sorta n numere reale.

35-2 Niveluri maximale
Fie Q o mulţime de n puncte în plan. Spunem că punctul (x, y) ***domină*** punctul (x', y') dacă $x \geq x'$ şi $y \geq y'$. Un punct din Q care nu este dominat de alte puncte din Q îl numim punct ***maximal***. Observăm că mulţimea Q poate să conţină mai multe puncte maximale, care pot fi organizate în ***niveluri maximale*** după cum urmează. Primul nivel maximal L_1 este mulţimea de puncte maximale din Q. Pentru $i > 1$, al i-lea nivel maximal L_i este mulţimea de puncte maximale din $Q - \bigcup_{j=1}^{i-1} L_j$.

Presupunem că mulţimea Q are k niveluri maximale nevide, şi fie y_i coordonata în y a celui mai din stânga punct din L_i pentru $i = 1, 2, ..., k$. Presupunem, de asemenea, că două puncte din Q nu au aceeaşi coordonată x sau y.

a. Arătaţi că $y_1 > y_2 > ... > y_k$.

Analizăm un punct (x, y) care este în stânga faţă de toate punctele din Q şi pentru care y este diferit faţă de coordonata y a oricărui punct din Q. Fie $Q' = Q \cup \{(x, y)\}$.

b. Fie j cel mai mic index, astfel încât $y_i < y$, excepţie $y < y_k$, caz în care considerăm $j = k + 1$. Arătaţi că nivelurile maximale ale lui Q' sunt după cum urmează.

- Dacă $j \leq k$, atunci nivelurile maximale ale lui Q' sunt aceleaşi cu nivelurile maximale ale lui Q, cu excepţia că L_j îl include şi pe (x, y) ca noul ei punct cel mai din stânga.
- Dacă $j = k + 1$, atunci primele k niveluri maximale ale lui Q' sunt aceleaşi cu ale lui Q, dar, în plus, Q' are un al $(k + 1)$-lea nivel maximal nevid: $L_{k+1} = (x, y)$.

c. Descrieţi un algoritm care calculează nivelurile maximale ale unei mulţimi Q de n puncte într-un timp $O(n \lg n)$. (*Indicaţie*: Deplasaţi o dreaptă de baleiere de la dreapta la stânga.)

d. Ce dificultăţi apar dacă permitem ca punctele de intrare să aibă aceeaşi coordonată x sau y? Propuneţi soluţii pentru aceste probleme.

35-3 Vânătorii de fantome şi fantomele

Un grup de n vânatori de fantome se războieşte cu n fantome. Fiecare vânător este înarmat cu o armă protonică, ce loveşte fantoma cu o rază, distrugând-o. Raza se propagă în linie dreaptă şi îşi încheie traiectoria în momentul în care nimereşte fantoma. Vânătorii de fantome îşi aleg una din următoarele strategii. Ei fac perechi cu fantomele, formând n perechi de vânători-fantome şi, în acest fel, simultan, fiecare vânător de fantome va elimina fantoma pereche. Precum se ştie, e foarte periculos ca două raze să se intersecteze şi, de aceea, vânătorii de fantome trebuie să-şi aleagă perechi pentru care nu apar intersecţii de raze.

Se presupune că poziţia fiecărui vânător de fantome şi a fiecărei fantome este un punct fix în plan şi că nu există trei puncte coliniare.

a. Demonstraţi că există o linie care trece printr-un vânător de fantome şi printr-o fantomă, astfel încât numărul vânătorilor de fantome aflaţi de o parte a liniei este egal cu numărul fantomelor aflate de aceeaşi parte. Descrieţi cum se poate determina o astfel de linie într-un timp $O(n \lg n)$.

b. Se cere un algoritm de timp $O(n^2 \lg n)$ care să echivaleze numărul vânătorilor de fantome cu numărul fantomelor în aşa fel, încât să nu se intesecteze nici o rază.

35-4 Distribuţie învelită rar

Analizăm problema calculării învelitorii convexe a unei mulţimi de puncte din plan care au fost obţinute conform unor distribuţii aleatoare cunoscute. Câteodată, învelitoarea convexă a n puncte care au fost obţinute dintr-o astfel de distribuţie, are dimensiunea $O(n^{1-\varepsilon})$ pentru o constantă $\varepsilon > 0$ oarecare. Numim aceasta o distribuţie ***învelită rar***. Distribuţiile învelite rar includ următoarele:

- Punctele au fost obţinute uniform dintr-o rază unitate a discului. Învelitoarea convexă are dimensiunea $\Theta(n^{1/3})$.

- Punctele au fost obţinute uniform din interiorul unui poligon convex având k laturi, oricare ar fi k constant. Învelitoarea convexă are dimensiunea $\Theta(\lg n)$.
- Punctele au fost obţinute după o distribuţie normală bidimensională. Învelitoarea convexă are dimensiunea $\Theta(\sqrt{\lg n})$.

a. Se dau două poligoane convexe cu n_1, respectiv n_2 vârfuri. Arătaţi cum se calculează învelitoarea convexă a celor $n_1 + n_2$ puncte într-un timp $O(n_1 + n_2)$. (Poligoanele se pot suprapune.)

b. Arătaţi că învelitoarea convexă a unei mulţimi de n puncte obţinute independent, conform unei distribuţii învelite rar, poate fi calculată într-un timp $O(n)$. (*Indicaţie:* Determinaţi recursiv învelitoarea convexă pentru primele $n/2$ puncte şi apoi pentru ultimele $n/2$ puncte şi apoi combinaţi rezultatele.)

Note bibliografice

Acest capitol este doar un punct de pornire pentru algoritmii şi tehnicile geometriei computaţionale. Dintre cărţile de geometrie computaţională amintim Preparata şi Shamos [160] şi Edelsbrunner [60].

Cu toate că geometria a fost studiată încă din antichitate, dezvoltarea de algoritmi pentru problemele geometrice este relativ nouă. Preparata şi Shamos afirmă că prima referire la complexitatea unei probleme a fost dată de E. Lemoine în 1902. El a studiat construcţiile euclidiene – cele care se pot realiza cu ajutorul riglei şi compasului în plan – şi a formulat cinci primitive: se pune un picior al compasului într-un punct dat, se pune un picior pe o dreaptă dată, se trasează cercul, se pune rigla într-un punct dat şi se trasează o dreaptă. Lemoine a studiat care este numărul de primitive necesar pentru a efectua o construcţie dată; el numeşte acest număr "simplitatea" construcţiei.

Algoritmul care determină dacă orice două segmente se intersectează, din secţiunea 35.2, se datorează lui Shamos şi Hoey [176].

Versiunea originară a scanării lui Graham este dată în Graham [91]. Algoritmul de împachetare este datorat lui Jarvis [112]. Folosind ca model un arbore de decizie, Yao [205] demonstrează că $\Omega(n \lg n)$ este marginea inferioară pentru timpul de execuţie al oricărui algoritm de determinare a învelitorii convexe. Când se ţine cont de numărul de vârfuri h ale învelitorii convexe, algoritmul trunchiază-şi-caută al lui Kirkpatrick şi Seidel [120], care necesită un timp $O(n \lg h)$, este asimptotic optimal.

Timpul de execuţie $O(n \lg n)$ necesar algoritmului divide şi stăpâneşte pentru determinarea celei mai apropiate perechi de puncte este dat de Shamos şi se găseşte în Preparata şi Shamos [160]. Preparata şi Shamos arată, de asemenea, că algoritmul este asimptotic optimal într-un model arbore de decizie.

36 NP-completitudine

Toţi algoritmii studiaţi până acum au fost ***algoritmi cu timp de execuţie polinomial:*** pentru date de intrare de mărime n, ordinul de complexitate în cel mai defavorabil caz este $O(n^k)$, unde k este o constantă. Este normal să ne punem întrebarea dacă *toate* problemele pot fi rezolvate în timp polinomial. Răspunsul este negativ. Există probleme care nu pot fi rezolvate de nici un calculator indiferent de timpul acordat pentru găsirea unei soluţii. Un exemplu, în acest sens, este faimoasa "problemă a şovăielii" propusă de Turing. Există, de asemenea, probleme care pot fi rezolvate, dar nu au ordinul de complexitate de forma $O(n^k)$, k constant. O problemă rezolvabilă în timp polinomial o vom numi problemă accesibilă, iar una al cărei ordin de complexitate depinde de o funcţie suprapolinomială o vom numi problemă inaccesibilă.

Subiectul acestui capitol îl constituie o clasă foarte interesantă de probleme, aşa-numitele probleme "NP-complete", al căror statut este necunoscut. Până în prezent, nu s-a descoperit nici un algoritm polinomial pentru rezolvarea unei astfel de probleme, dar nimeni nu a reuşit să demonstreze că există o limită inferioară pentru timpul suprapolinomial care este necesar rezolvării acesteia. Această aşa-numită problemă P $\neq$ NP reprezintă, încă de la formularea ei în 1971, unul dintre cele mai interesante subiecte deschise în algoritmica teoretică.

Cei mai mulţi informaticieni cred că problemele NP-complete sunt inaccesibile. Explicaţia este că, dacă se descoperă o rezolvare polinomială pentru o singură problemă NP-completă, atunci, poate fi găsit un algoritm polinomial pentru rezolvarea *oricărei* alte probleme din această clasă. Datorită vastei arii pe care o acoperă problemele NP-complete care au fost studiate până acum fără să fi apărut nici un progres în direcţia găsirii unei rezolvări polinomiale pentru vreuna dintre ele, ar fi într-adevăr uluitor ca toate să poată fi totuşi rezolvate în timp polinomial.

Pentru a deveni un bun programator, trebuie să înţelegeţi noţiunile de bază ale teoriei NP-completitudinii. Pentru a afirma că o problemă este NP-completă, trebuie dată o demonstraţie a inaccesibilităţii sale. Pentru un inginer, ar fi mai bine să-şi petreacă timpul dezvoltând un algoritm de aproximare (vezi capitolul 37) decât să caute un algoritm rapid care să dea o soluţie exactă. Mai mult, un număr mare de probleme obişnuite şi interesante, care la prima vedere nu par a fi mai dificile decât o sortare, o căutare în grafuri sau un flux în reţea, sunt, de fapt, NP-complete. Aşasar este important să vă familiarizaţi cu această clasă remarcabilă de probleme.

În acest capitol vor fi studiate aspecte ale NP-completitudinii care se referă la analiza algoritmilor. În secţiunea 36.1 vom da o definiţie a noţiunii de "problemă", vom defini clasa de complexitate P a problemelor de decizie rezolvabile în timp polinomial. Vom vedea, de asemenea, care este locul acestor noţiuni în cadrul teoriei limbajelor formale. În secţiunea 36.2 se defineşte clasa NP a problemelor de decizie ale căror soluţii pot fi verificate în timp polinomial. Tot aici, vom pune şi întrebarea P $\neq$ NP.

În secţiunea 36.3 se arată cum pot fi studiate legăturile dintre probleme prin intermediul "reducerilor" în timp polinomial. Este definită NP-completitudinea şi se schiţează o demonstraţie a faptului că o problemă, numită "problema satisfiabilităţii circuitului", este NP-completă. După ce am demonstrat inaccesibilitatea unei probleme, în secţiunea 36.4 vom arăta cum poate fi demonstrat faptul că o problemă este NP-completă folosind metoda reducerii. Metodologia este ilustrată prin demonstrarea faptului că alte două probleme de satisfiabilitate sunt NP-complete. În secţiunea 36.5 se demonstrează NP-completitudinea altor probleme.

36.1. Timpul polinomial

Începem studiul NP-completitudinii definind noţiunea de problemă rezolvabilă în timp polinomial. Aceste probleme sunt, în general, privite ca fiind accesibile, din motive mai mult filosofice decât matematice. Putem oferi trei argumente pentru ca problemele rezolvabile în timp polinomial să fie considerate accesibile.

În primul rând, chiar dacă ar fi rezonabil să considerăm o problemă cu ordinul de complexitate $\Theta(n^{100})$ ca fiind inaccesibilă, există foarte puţine probleme practice rezolvabile într-un timp care să depindă de o funcţie polinomială al cărei grad este atât de mare. Problemele rezolvabile în timp polinomial întâlnite în practică necesită, în marea majoritate a cazurilor, mult mai puţin timp pentru a fi rezolvate.

În al doilea rând, pentru multe modele de calcul, o problemă care poate fi rezolvată în timp polinomial folosind un model de calcul, va putea fi rezolvată în timp polinomial folosind orice alt model de calcul. De exemplu, clasa problemelor rezolvabile în timp polinomial cu ajutorul maşinilor seriale cu acces direct coincide cu clasa problemelor rezolvabile în timp polinomial cu ajutorul maşinilor virtuale Turing.[1] Această clasă coincide, de asemenea, cu clasa problemelor rezolvabile în timp polinomial folosind un calculator care are posibilitatea procesării paralele, chiar dacă numărul procesoarelor creşte după o funcţie polinomială care depinde de mărimea intrării.

În al treilea rând, clasa problemelor rezolvabile în timp polinomial are câteva proprietăţi de închidere interesante, deoarece polinoamele sunt mulţimi închise în cazul adunării, înmulţirii şi al compunerii. De exemplu, dacă ieşirea unui algoritm polinomial este considerată intrare pentru un alt algoritm polinomial, atunci algoritmul rezultat va fi unul polinomial. De asemenea, dacă un algoritm polinomial apelează de un număr constant de ori subrutine polinomiale, atunci, algoritmul rezultat este polinomial.

Probleme abstracte

Pentru a înţelege clasa problemelor rezolvabile în timp polinomial trebuie, pentru început, să definim noţiunea de "problemă". Definim o ***problemă abstractă*** Q ca fiind a relaţie binară între mulţimea I a ***instanţelor*** problemei şi mulţimea S a ***soluţiilor*** problemei. De exemplu, să considerăm problema DRUM-MINIM care cere determinarea celui mai scurt drum dintre două noduri ale unui graf neorientat fără costuri, $G = (V, E)$. O instanţă a acestei probleme este un triplet format dintr-un graf şi două noduri. O soluţie este o secvenţă de noduri ale grafului sau, probabil, o secvenţă vidă în cazul în care nu există drum între cele două noduri. Însăşi problema DRUM-MINIM este relaţia care asociază fiecărei instanţe formate dintr-un graf şi două noduri o soluţie care precizează cel mai scurt drum care uneşte cele două noduri. Deoarece, drumurile minime nu sunt, în mod obligatoriu, unice, o anumită instanţă dată a problemei poate avea mai multe soluţii.

Această formulare a definiţiei unei probleme abstracte este mai generală decât este nevoie pentru ca scopul acestei cărţi să fie atins. Pentru a simplifica lucrurile, teoria NP-completitudinii se referă numai la ***problemele de decizie:*** cele care au soluţii de tipul da/nu. În acest caz, putem privi o problemă abstractă de decizie ca fiind o funcţie având ca domeniu mulţimea I a

[1] vezi Hopcroft şi Ullman [104] sau Lewis şi Papadimitriou [139] pentru detalii despre modelul maşinii Turing.

instanţelor problemelor, codomeniul fiind mulţimea $\{0, 1\}$. De exemplu, o problemă de decizie DRUM care se referă la drumul minim într-un graf ar putea fi formulată astfel: "Dându-se un graf $G = (V, E)$, două vârfuri $u, v \in V$ şi un număr întreg nenegativ k, aflaţi dacă există un drum în G, între u şi v, a cărui lungime este cel mult k." Dacă $i = \langle G, u, v, k\rangle$ este o instanţă a acestei probleme, atunci DRUM$(i) = 1$ (da), dacă drumul minim dintre u şi v are cel mult lungimea k şi DRUM$(i) = 0$ (nu), în caz contrar.

Multe probleme abstracte nu sunt probleme de decizie, ci, în cele mai multe cazuri, ***probleme de optim***, caz în care o anumită valoare trebuie să fie minimizată sau maximizată. Pentru a putea aplica teoria NP-completitudinii în cazul problemelor de optim, trebuie să le transformăm în probleme de decizie. Modalitatea uzuală de a transforma o problemă de optim într-una de decizie este impunerea, pentru valoarea care trebuie optimizată, a unei limite inferioare, în cazul problemelor de maximizare, sau a uneia superioare, în cazul problemelor de minimizare. Un exemplu ar fi transformarea problemei drumului minim în problema de decizie DRUM prin impunerea limitei superioare k.

Cu toate că teoria NP-completitudinii ne obligă să transformăm problemele de optim în probleme de decizie, aceasta nu diminuează importanţa pe care o are această teorie. În general, dacă putem rezolva rapid o problemă de optim, atunci, vom putea rezolva problema de decizie corespunzătoare la fel de rapid. Va trebui doar să impunem, ca limită a valorii care trebuie optimizată, exact valoarea obţinută în urma rezolvării problemei de optim. Dacă o problemă de optim este uşor de rezolvat, atunci şi problema de decizie corespunzătoare este uşor de rezolvat. Această afirmaţie, exprimată într-un mod care are o relevanţă pentru NP-completitudine, devine: dacă se poate demonstra că o problemă de decizie este greu de rezolvat, atunci se va putea demonstra şi că problema de optim corespunzătoare este greu de rezolvat. Aşadar, chiar dacă teoria NP-completitudinii îşi limitează atenţia la problemele de decizie, aria ei de aplicabilitate este mult mai mare.

Codificări

Dacă un program trebuie să rezolve o problemă abstractă, atunci, instanţele problemei trebuie să poată fi reprezentate astfel încât programul să fie capabil să le înţeleagă. O ***codificare*** a unei mulţimi S de obiecte abstracte este o funcţie e, de la S la mulţimea şirurilor binare.[2] De exemplu, o codificare a mulţimii numerelor naturale $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ este mulţimea de şiruri $\{0, 1, 10, 11, 100, \ldots\}$. Folosind această codificare, obţinem $e(17) = 10001$. Cei care au aruncat o privire asupra reprezentărilor caracterelor de pe tastatură sunt familiarizaţi cu unul din codurile ASCII sau EBCDIC. În cod ASCII, avem $e(\text{A}) = 1000001$. Chiar şi un obiect compus poate fi codificat folosind şiruri prin combinarea reprezentărilor părţilor componente. Poligoanele, grafurile, funcţiile, perechile ordonate sau programele pot fi toate codificate ca şiruri binare.

Aşadar, un algoritm care "rezolvă" probleme de decizie abstracte cu ajutorul unui calculator, are ca intrare o codificare a instanţei problemei. Vom numi ***problemă concretă*** o problemă a cărei mulţime a instanţelor este o mulţime de şiruri binare. Spunem că un algoritm ***rezolvă*** o problemă concretă în timp $O(T(n))$ dacă, atunci când este dată o instanţă i de lungime $n = |i|$, algoritmul va găsi o soluţie cel mult într-un timp $O(T(n))$.[3]

[2] Codomeniul lui e nu trebuie să conţină şiruri *binare*, orice mulţime de şiruri peste un alfabet finit, având cel puţin două simboluri, fiind potrivită.

[3] Presupunem că ieşirea algoritmului este separată de intrare. Deoarece este nevoie de cel puţin un pas pentru determinarea fiecărui bit al ieşirii şi există $O(T(n))$ paşi, mărimea ieşirii este $O(T(n))$. O problemă concretă este

Acum putem defini ***clasa de complexitate*** **P** ca fiind mulţimea problemelor concrete de decizie care sunt rezolvabile în timp polinomial.

Putem folosi codificări pentru a stabili o corespondenţă între problemele abstracte şi problemele concrete. Dându-se o problemă de decizie abstractă Q care stabileşte o funcţie de la mulţimea I a instanţelor la mulţimea $\{0, 1\}$, o codificare $e : I \to \{0,1\}^*$ poate fi folosită pentru a găsi problema concretă de decizie corespunzătoare pe care o vom nota cu $e(Q)$. Dacă soluţia unei instanţe $i \in I$ a problemei abstracte este $Q(i) \in \{0,1\}$, atunci, soluţia instanţei $e(i) \in \{0,1\}^*$ este, de asemenea, $Q(i)$. Ca un detaliu tehnic, putem aminti faptul că pot exista unele şiruri binare care nu reprezintă o instanţă a problemei abstracte. Pentru comoditate, vom considera că funcţia-problemă asociază unui astfel de şir valoarea 0. Aşadar o problemă concretă are aceleaşi soluţii ca şi problema abstractă corespunzătoare, dacă instanţele formate din şiruri binare reprezintă codificări ale instanţelor problemei abstracte.

Am vrea să extindem definiţia rezolvabilităţii în timp polinomial de la problemele concrete la cele abstracte folosind codificarea ca o punte de legătură între aceste tipuri de probleme, dar am dori, totodată, ca definiţia să nu depindă de vreo codificare particulară. Aceasta înseamnă că eficienţa cu care este rezolvată o problemă nu ar trebui să depindă de codificarea aleasă. Din nefericire, această eficienţă depinde însă destul de mult de modul în care sunt codificate datele. De exemplu, să presupunem că intrarea pentru un anumit algoritm este numărul întreg k şi că timpul de execuţie este $\Theta(k)$. Dacă numărul întreg k va fi reprezentat ***unar***, atunci, ordinul de complexitate va fi $O(n)$ unde n este lungimea intrării. Dacă numărul întreg va fi reprezentat în baza 10, atunci, lungimea intrării va fi $n = \lfloor \lg k \rfloor + 1$. În acest caz, ordinul de complexitate va fi $\Theta(k) = \Theta(2^n)$, adică un timp exponenţial faţă de lungimea intrării. Deci, în funcţie de codificarea aleasă, programul care va implementa algoritmul se va executa fie în timp polinomial, fie în timp suprapolinomial.

Codificarea unei probleme abstracte are o importanţă deosebită pentru înţelegerea noţiunii de timp polinomial. Nu putem vorbi despre rezolvarea unei probleme abstracte fără să specificăm mai întâi care va fi codificarea acesteia. Cu toate acestea, în practică, dacă excludem codificările "costisitoare" cum ar fi cele unare, codificarea unei probleme nu are o prea mare influenţă asupra complexităţii algoritmului. De exemplu, în cazul în care întregii sunt reprezentaţi în baza 3 în loc de baza 2, timpul de execuţie va rămâne polinomial şi pentru baza 3 dacă a fost polinomial pentru baza 2, deoarece un întreg în baza 3 poate fi convertit în baza 2 într-un timp polinomial.

Spunem că o funcţie $f : \{0,1\}^* \to \{0,1\}^*$ este ***calculabilă în timp polinomial*** dacă există un algoritm în timp polinomial A, pentru care, dându-se o intrare $x \in \{0,1\}^*$, ieşirea produsă este $f(x)$. Pentru o mulţime I a instanţelor unei probleme, spunem că două codificări e_1 şi e_2 sunt ***asociate poliomial*** dacă există două funcţii calculabile în timp polinomial f_{12} şi f_{21} astfel încât, pentru orice $i \in I$, să fie satisfăcute relaţiile $f_{12}(e_1(i)) = e_2(i)$ şi $f_{21}(e_2(i)) = e_1(i)$. Aceasta ar însemna că una dintre codificări poate fi determinată cu ajutorul celeilalte în timp polinomial. Dacă două codificări e_1 şi e_2 ale unei probleme abstracte sunt asociate polinomial, atunci nu are nici o importanţă care dintre ele este folosită, algoritmul fiind sau polinomial pentru ambele codificări, sau suprapolinomial. Acest rezultat este demonstrat în lema următoare.

Lema 36.1 Fie Q o problemă abstractă de decizie, I mulţimea instanţelor sale, şi e_1 şi e_2 două codificări pentru I asociate polinomial. Atunci $e_1(Q) \in P$ dacă şi numai dacă $e_2(Q) \in P$.

---

rezolvabilă în timp polinomial dacă există un algoritm care găseşte o rezolvare într-un timp de ordinul $O(n^k)$, k fiind o constantă.

Demonstraţie. Este nevoie să demonstrăm implicaţia doar într-un singur sens deoarece implicaţiile sunt simetrice. Să presupunem deci, că $e_1(Q)$ poate fi rezolvată în timp $O(n^k)$, cu k constant. Presupunem, totodată, că, pentru o instanţă i, codificarea $e_1(i)$ poate fi determinată din codificarea $e_2(i)$ într-un timp $O(n^c)$, cu c constant, unde $n = |e_2(i)|$. Pentru a rezolva problema folosind codificarea $e_2(Q)$, pentru o intrare $e_2(i)$, calculăm mai întâi $e_1(i)$ şi apoi rulăm algoritmul pentru $e_1(Q)$ şi intrarea $e_1(i)$. Să calculăm acum timpul necesar acestor operaţii. Pentru schimbarea codificării avem nevoie de un timp $O(n^c)$, aşadar $|e_1(i)| = O(n^c)$, deoarece ieşirea unui calculator serial nu poate fi mai mare decât timpul său de execuţie. Rezolvarea problemei pentru $e_1(i)$ poate fi făcută într-un timp $O(|e_1(i)|^k) = O(n^{ck})$, ceea ce reprezintă un timp polinomial deoarece atât c, cât şi k, sunt constante. ∎

Aşadar, nu contează dacă instanţele unei probleme sunt codificate în baza 2 sau în baza 3, "complexitatea" problemei nefiind afectată de acest fapt, adică va fi polinomială sau suprapolinomială în ambele situaţii. Totuşi, dacă instanţele sunt codificate unar, complexitatea algoritmului s-ar putea schimba. Pentru a putea face conversii într-un mod independent de codificare, vom presupune, în general, că instanţele problemelor sunt codificate într-o manieră rezonabilă şi concisă, dacă nu este specificat contrariul. Pentru a fi mai exacţi, vom presupune că o codificare aleasă pentru un întreg este asociată polinomial cu reprezentarea în baza 2 a întregului, şi că o codificare a unei mulţimi finite este asociată polinomial cu codificarea ei ca listă a elementelor componente, cuprinse între acolade şi separate prin virgule. (ASCII este o astfel de schemă de codificare.) Având la dispoziţie un astfel de "standard" pentru codificări, putem obţine codificări rezonabile pentru alte obiecte matematice cum ar fi tuplele, grafurile şi formulele. Pentru a simboliza codificarea unui obiect, vom scrie simbolul obiectului între paranteze unghiulare. Deci, $\langle G \rangle$ va simboliza codificarea standard a grafului G.

Atât timp cât folosim, în mod implicit, o codificare care este asociată polinomial cu o codificare standard, putem vorbi, direct, despre probleme abstracte fără a mai face referire la codificări particulare, ştiind că alegerea unei anumite codificări nu are nici un efect asupra complexităţii algoritmului. De aici înainte vom presupune că, în general, instanţele unei probleme sunt codificate prin şiruri binare, folosind codificarea standard, dacă nu se precizează contrariul în mod explicit. De asemenea, vom neglija diferenţa dintre problemele abstracte şi problemele concrete. Cititorul ar trebui să fie atent la problemele practice pentru care codificarea standard nu este evidentă şi pentru care alegerea unei anumite codificări are efecte asupra complexităţii algoritmului.

Limbaje formale

Unul dintre aspectele cele mai avantajoase în rezolvarea problemelor de decizie îl reprezintă faptul că sunt foarte uşor de folosit uneltele oferite de teoria limbajelor formale. Merită, deci, să recapitulăm câteva definiţii care apar în cadrul acestei teorii. Un ***alfabet*** Σ este o mulţime finită de simboluri. Un ***limbaj*** L peste Σ este orice mulţime de şiruri formate din simboluri din Σ. De exemplu, dacă $\Sigma = \{0, 1\}$, atunci, mulţimea $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ este limbajul reprezentării binare a numerelor prime. Simbolizăm ***şirul vid*** prin ε şi ***limbajul vid*** prin $\emptyset$. Limbajul tuturor şirurilor peste Σ este notat prin Σ^*. De exemplu, dacă $\Sigma = \{0, 1\}$, atunci, $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ este mulţimea tuturor şirurilor binare. Orice limbaj L peste Σ este o submulţime a lui Σ^*.

Există o varietate mare de operaţii cu limbaje. Operaţii din teoria mulţimilor cum ar

fi ***reuniunea*** şi ***intersecţia*** rezultă, direct, din definirea teoretică a mulţimilor. Definim ***complementul*** lui L prin $\overline{L} = \Sigma^* - L$. ***Concatenarea*** a două limbaje L_1 şi L_2 este limbajul

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ şi } x_2 \in L_2\}.$$

Închiderea sau ***steaua Kleene*** a unui limbaj L este limbajul

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots,$$

unde L^k este limbajul obţinut prin concatenarea limbajului L cu el însuşi de k ori.

Din punct de vedere al teoriei limbajelor, mulţimea instanţelor unei probleme Q este mulţimea Σ^*, unde $\Sigma = \{0, 1\}$. Datorită faptului că problema Q este total caracterizată prin acele instanţe care duc la obţinerea rezultatului 1 (da), putem privi mulţimea Q ca fiind un limbaj L peste $\Sigma = \{0, 1\}$, unde

$$L = \{x \in \Sigma^* : Q(x) = 1\}.$$

De exemplu, problema de decizie DRUM are ca limbaj corespunzător limbajul

$$\begin{aligned} \text{DRUM} = \{\langle G, u, v, k\rangle \quad : \quad & G = (V, E) \text{ este un graf neorientat, } u, v \in V, \\ & k \geq 0 \text{ este un întreg, şi} \\ & \text{există un drum de la } u \text{ la } v \text{ în } G \text{ a cărui lungime este cel mult } k\}. \end{aligned}$$

(Unde este convenabil vom folosi acelaşi nume – DRUM în acest caz – pentru a ne referi atât la problema de decizie cât şi la limbajul corespunzător.)

Limbajele formale ne permit să exprimăm relaţii între problemele de decizie şi anumiţi algoritmi care le rezolvă. Spunem că un algoritm A ***acceptă*** un şir $x \in \{0, 1\}^*$ dacă, dându-se o intrare x, ieşirea algoritmului este $A(x) = 1$. Limbajul ***acceptat*** de un algoritm A este mulţimea $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, adică mulţimea şirurilor pe care algoritmul le acceptă. Un algoritm A ***respinge*** un şir x dacă $A(x) = 0$.

Chiar dacă limbajul L este acceptat de un algoritm A, algoritmul nu va respinge neapărat un şir $x \notin L$ care îi este dat ca intrare. De exemplu algoritmul ar putea intra în ciclu infinit. Un limbaj L este ***clarificat*** de un algoritm A dacă fiecare şir binar din L este acceptat de către A şi fiecare şir binar care nu este în L este respins de către A. Un limbaj L este ***acceptat în timp polinomial*** de către un algoritm A dacă există o constantă k astfel încât, pentru orice şir binar x de lungime n din L, algoritmul îl acceptă pe x în timp $O(n^k)$. Un limbaj L este ***clarificat în timp polinomial*** de către un algoritm A dacă, pentru orice şir $x \in \{0, 1\}^*$ de lungime n, algoritmul decide în mod corect dacă $x \in L$ într-un timp $O(n^k)$ pentru o constantă k. Deci, pentru a accepta un limbaj, un algoritm trebuie să trateze numai şirurile din L, dar pentru a clarifica un limbaj el trebuie să accepte sau să respingă în mod corect orice şir din $\{0, 1\}^*$.

De exemplu, limbajul DRUM poate fi acceptat în timp polinomial. Un algoritm polinomial care acceptă limbajul DRUM determină cel mai scurt drum din G, dintre u şi v, folosind căutarea în lăţime şi apoi compară distanţa obţinută cu k. Dacă distanţa este cel mult k atunci algoritmul furnizează ieşirea 1 şi se opreşte. În caz contrar, algoritmul va intra în ciclu infinit. Acest algoritm nu clarifică DRUM deoarece nu furnizează ieşirea 0 în cazul în care cel mai scurt drum are o lungime mai mare decât k. Un algoritm care clarifică DRUM trebuie să respingă în mod explicit şirurile binare care nu fac parte din DRUM. Pentru o problemă de decizie cum este DRUM, un

algoritm care clarifică limbajul DRUM este uşor de proiectat. Pentru alte probleme cum ar fi problema şovăielii a lui Turing există un algoritm de acceptare, dar nu există unul de clarificare.

Putem defini ***clasa de complexitate*** ca fiind o mulţime de limbaje pentru care apartenenţa unui element este determinată de o ***măsură a complexităţii*** unui algoritm (cum ar fi timpul de execuţie) care determină dacă un şir dat x aparţine sau nu limbajului L. Definiţia riguroasă a clasei de complexitate este una mult mai tehnică – cititorii interesaţi pot consulta o lucrare ai cărei autori sunt Hartmanis şi Stearns [95].

Folosind teoria limbajelor formale, putem da o definiţie alternativă a clasei de complexitate P:

$$\mathrm{P} = \{L \subseteq \{0,1\}^* \quad : \quad \text{există un algoritm } A \text{ care clarifică } L \text{ în timp polinomial}\}.$$

De fapt, P este şi clasa limbajelor care pot fi acceptate în timp poliomial.

Teorema 36.2 P = $\{L$: L este acceptat de un algoritm care se execută în timp polinomial$\}$.

Demonstraţie. Deoarece clasa limbajelor care pot fi clarificate în timp polinomial de către un algoritm este o submulţime a clasei limbajelor care pot fi acceptate în timp polinomial de către un algoritm, este suficient să arătăm că, dacă L este acceptat de un algoritm polinomial, el poate fi clarificat într-un timp polinomial. Fie L un limbaj acceptat în timp polinomial de către un algoritm A. Vom da o demonstraţie clasică de "simulare" pentru a construi un alt algoritm A' care clarifică L. Datorită faptului că A acceptă L într-un timp $O(n^k)$, cu k constant, există o constantă c astfel încât A acceptă L în cel mult $T = cn^k$ paşi. Pentru fiecare şir x furnizat la intrare, algoritmul A' simulează acţiunile realizate de algoritmul A pentru un timp T. La sfârşitul acestui interval de timp T, algoritmul A' verifică algoritmul A. Dacă A l-a acceptat pe x, A' îl acceptă pe x şi furnizează ieşirea 1. Dacă A nu l-a acceptat pe x, A' îl respinge pe x şi furnizează ieşirea 0. Faptul că A' simulează A nu creşte timpul de execuţie decât, cel mult, cu un factor polinomial, deci A' este un algoritm polinomial care îl clarifică pe L. ∎

Trebuie remarcat faptul că demonstraţia teoremei 36.2 este una neconstructivă. Pentru un limbaj $L \in P$, s-ar putea să nu cunoaştem o limită pentru timpul de execuţie al algoritmului A care îl acceptă pe L. Totuşi, ştim că o astfel de limită există şi, ca urmare, există şi un algoritm A' care verifică existenţa acestei limite, chiar dacă găsirea algoritmului A' nu este o sarcină uşoară.

Exerciţii

36.1-1 Definiţi problema de optimizare LUNGIMEA-CELUI-MAI-LUNG-DRUM ca o relaţie ce asociază fiecare instanţă a unui graf neorientat şi a două noduri cu lungimea celui mai lung drum elementar dintre cele două noduri. Definiţi problema de decizie CEL-MAI-LUNG-DRUM = $\{\langle G, u, v, k\rangle : G = (V, E)$ este un graf neorientat, $u, v, \in V, k \geq 0$ este un număr întreg, şi există un drum elementar de la u la v în G a cărui lungime este cel puţin $k\}$. Arătaţi că problema LUNGIMEA-CELUI-MAI-LUNG-DRUM poate fi rezolvată în timp polinomial dacă şi numai dacă, CEL-MAI-LUNG-DRUM ∈ P.

36.1-2 Daţi o definiţie pentru problema găsirii celui mai lung ciclu elementar într-un graf neorientat. Formulaţi problema de decizie corespunzătoare. Precizaţi care este limbajul corespunzător problemei de decizie.

36.1-3 Realizaţi o codificare pentru grafurile orientate ca şiruri binare folosind reprezentarea prin matrice de adiacenţă. Realizaţi acelaşi lucru folosind reprezentarea prin liste de adiacenţă. Demonstraţi că cele două codificări sunt asociate polinomial.

36.1-4 Este algoritmul de programare dinamică a problemei 0-1 a rucsacului cerut la exericţiul 17.2-2 un algoritm polinomial? Justificaţi răspunsul.

36.1-5 Să presupunem că există un limbaj L pentru care există un algoritm care acceptă orice şir $x \in L$ în timp polinomial, dar acest algoritm se execută în timp suprapolinomial dacă $x \notin L$. Demonstraţi că L poate fi clarificat în timp polinomial.

36.1-6 Demonstraţi că un algoritm care face cel mult un număr constant de apeluri la subrutine polinomiale se execută în timp polinomial, dar că un număr polinomial de apeluri poate duce la un algoritm care se execută în timp exponenţial.

36.1-7 Demonstraţi afirmaţia: clasa P, văzută ca o mulţime de limbaje, este închisă pentru operatorii de reuniune, intersecţie, concatenare, complementare sau steaua lui Kleene. Adică, dacă $L_1, L_2 \in \mathrm{P}$, atunci $L_1 \cup L_2 \in \mathrm{P}$, etc.

36.2. Verificări în timp polinomial

Vom studia, în continuare, algoritmii care "verifică" apartenenţa la diferite limbaje. De exemplu, să presupunem că, pentru o instanţă dată $\langle G, u, v, k\rangle$ a problemei de decizie DRUM, avem dat şi un drum d de la u la v. Putem verifica foarte uşor dacă lungimea lui d este cel mult k şi, dacă este aşa, putem privi d ca o "probă" a faptului că această instanţă aparţine într-adevăr mulţimii DRUM. Pentru problema de decizie DRUM această probă nu ne ajută prea mult. Până la urmă, DRUM aparţine clasei P – de fapt, DRUM poate fi rezolvată în timp liniar – aşadar verificarea apartenenţei pentru d ia tot atâta timp cât ar fi necesar pentru rezolvarea problemei iniţiale. Vom studia acum o problemă pentru care nu se cunoaşte încă un algoritm polinomial de clarificare, dar pentru care, dându-se o instanţă ca probă, verificarea este foarte uşoară.

Cicluri hamiltoniene

Problema găsirii unui ciclu hamiltonian într-un graf neorientat a fost studiată încă de acum o sută de ani. Un ***ciclu hamiltonian*** al unui graf neorientat $G = (V, E)$ este un ciclu elementar care conţine fiecare nod din V. Un graf care conţine un ciclu hamiltonian este numit ***graf hamiltonian***; în caz contrar graful este numit ***nehamiltonian***. Bondy şi Murty [31] citează o scrisoare a lui W. R. Hamilton în care este descris un joc matematic cu un dodecaedru (figura 36.1(a)) în care un jucător fixează cinci ace în oricare cinci vârfuri consecutive, şi celălalt trebuie să completeze drumul pentru a forma un ciclu ce conţine toate vârfurile. Dodecaedrul este hamiltonian, iar figura 36.1(a) arată un ciclu hamiltonian. Totuşi, nu toate grafurile sunt hamitoniene. De exemplu, figura 36.1(b) arată un graf bipartit cu un număr impar de vârfuri. (Exerciţiul 36.2-2 cere să se demonstreze că toate grafurile de acest tip nu sunt hamiltoniene.)

Putem defini ***problema ciclului hamiltonian***, "Conţine graful G un ciclu hamiltonian?" ca un limbaj formal:

$$\text{CICLU-HAM} = \{\langle G\rangle : G \text{ este graf hamiltonian}\}.$$

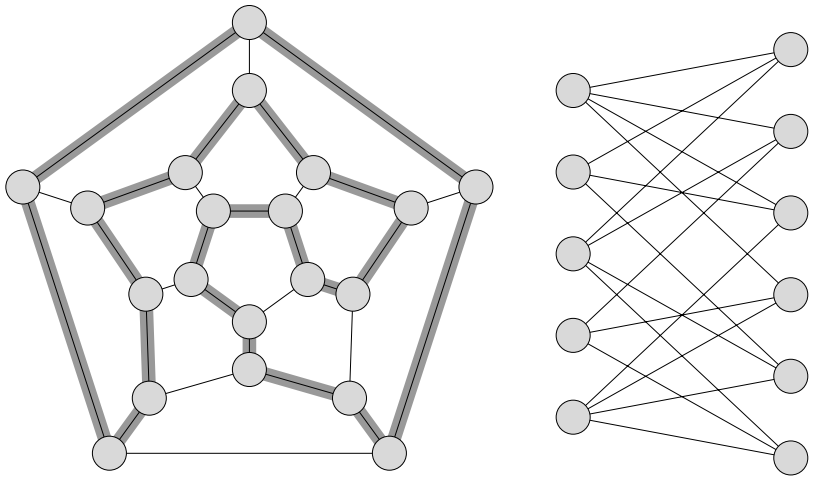


Figura 36.1 **(a)** Un graf reprezentând vârfurile, laturile şi feţele unui dodecaedru, cu un ciclu hamiltonian vizualizat prin laturi îngroşate. **(b)** Un graf bipartit cu un număr impar de vârfuri. Orice astfel de graf nu este hamiltonian.

Cum ar putea un algoritm să clarifice limbajul CICLU-HAM? Dându-se o instanţă $\langle G \rangle$ a problemei, un posibil algoritm de clarificare ar determina toate permutările vârfurilor lui G şi apoi ar verifica, pentru fiecare permutare, dacă este un ciclu hamiltonian. Care este timpul de execuţie pentru acest algoritm? Dacă folosim o codificare "rezonabilă" pentru graf, cum ar fi matricea de adiacenţă, atunci numărul m al vârfurilor din graf este $\Omega(\sqrt{n})$, unde $n = |\langle G \rangle|$ este lungimea codificării lui G. Există $m!$ permutări posibile ale vârfurilor, deci timpul de execuţie va fi $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$ care nu este de forma $O(n^k)$, pentru k constant. Deci, acest algoritm naiv nu se execută în timp polinomial şi, de fapt, problema ciclului hamiltonian este o problemă NP-completă, după cum se va dovedi în secţiunea 36.5.

Algoritmi de verificare

Să considerăm o problemă puţin mai simplă. Să presupunem că un prieten vă spune că un graf G este hamiltonian şi se oferă să demonstreze acest lucru precizând ordinea în care apar vârfurile în ciclul hamiltonian. Cu siguranţă, verificarea demonstraţiei va fi o sarcină destul de uşoară: se verifică pur şi simplu că ciclul furnizat este hamiltonian, verificând dacă acesta reprezintă o permutare a vârfurilor grafului şi dacă fiecare din muchiile ciclului există în graf. Acest algoritm de verificare poate fi cu siguranţă implementat astfel încât să se execute într-un timp $O(n^2)$, unde n este lungimea codificării lui G. Deci, o demonstraţie a faptului că există un ciclu hamiltonian într-un graf poate fi verificată în timp polinomial.

Definim un ***algoritm de verificare*** ca fiind un algoritm A cu doi parametri, unul fiind intrarea obişnuită reprezentată de un şir x, iar al doilea fiind un alt şir binar y numit ***probă***. Un astfel de algoritm ***verifică*** o intrare x dacă există o probă y astfel încât $A(x, y) = 1$. ***Limbajul***

verificat de un algoritm de verificare A este

$$L = \{x \in \{0,1\}^* : \text{ există } y \in \{0,1\}^* \text{ astfel încât } A(x,y) = 1\}.$$

Intuitiv un algoritm A verifică un limbaj L dacă pentru orice şir $x \in L$, există o probă y pe care A o poate folosi pentru a demonstra că $x \in L$. Mai mult, pentru orice şir $x \notin L$, nu este permisă existenţa unei probe care să dovedească faptul că $x \in L$. De exemplu, în problema ciclului hamiltonian, proba este lista vârfurilor din ciclul hamiltonian. Dacă un graf este hamiltonian atunci ciclul hamiltonian însuşi oferă destule informaţii pentru ca acest fapt să poată fi verificat. Invers, dacă un graf nu este hamiltonian, nu există nici o listă de vârfuri care ar putea să înşele algoritmul de verificare făcându-l să creadă că un astfel de graf este hamiltonian, deoarece algoritmul de verificare analizează cu atenţie ciclul propus pentru a fi sigur că acesta este hamiltonian.

Clasa de complexitate NP

Clasa de complexitate **NP** este clasa limbajelor care pot fi verificate de un algoritm în timp polinomial.[4] Mai precis, un limbaj L aparţine clasei NP, dacă şi numai dacă există un algoritm polinomial A cu două intrări şi o constantă c astfel încât

$$L = \{x \in \{0,1\}^* \quad : \quad \text{există o probă } y \text{ cu } |y| = O(|x|^c) \text{ astfel încât } A(x,y) = 1\}.$$

Spunem că algoritmul A ***verifică*** limbajul L ***în timp polinomial***.

Din discuţia anterioară privind ciclul hamiltonian rezultă că Ciclu-Ham $\in$ NP. (Este întotdeauna îmbucurător să ştim că o mulţime importantă nu este vidă.) Mai mult, dacă $L \in$ P, atunci $L \in$ NP, deoarece, dacă există un algoritm polinomial care clarifică L, acest algoritm poate fi foarte uşor transformat într-unul cu doi parametri care ignoră pur şi simplu orice probă şi acceptă exact acele intrări pe care le determină ca aparţinând lui L. Având în vedere cele de mai sus, putem trage concluzia că P $\subseteq$ NP.

Nu se ştie dacă P = NP, dar cei mai mulţi cercetători cred că P şi NP nu reprezintă aceeaşi clasă. Intuitiv, clasa P constă din problemele care pot fi rezolvate rapid. Clasa NP constă din problemele a căror soluţie poate fi verificată rapid. S-ar putea ca, din experienţă, să ştiţi deja că, de obicei, este mult mai greu să rezolvi o problemă decât să verifici dacă o anumită soluţie dată este corectă. Informaticienii cred, în general, că această analogie se extinde asupra claselor P şi NP şi, ca urmare, NP include limbaje care nu apar în P.

Există o altă probă atrăgătoare care ne-ar face să credem că P $\neq$ NP şi anume existenţa limbajelor "NP-complete". Vom studia această clasă în secţiunea 36.3.

Multe alte întrebări fundamentale pe lângă P $\neq$ NP rămân fără răspuns. În ciuda muncii depuse de mulţi cercetători, nimeni nu ştie dacă clasa NP este închisă faţă de operatorul complement, respectiv, nu s-a răspuns încă la întrebarea: $L \in$ NP implică $\overline{L} \in$ NP? Putem defini ***clasa de complexitate*** **co-NP** ca fiind mulţimea limbajelor L pentru care $\overline{L} \in$ NP. Întrebarea dacă NP este închisă faţă de operatorul complement poate fi reformulată astfel: sunt mulţimile NP şi co-NP identice? Deoarece clasa P este închisă faţă de complement (exerciţiul 36.1-7), rezultă că P $\subseteq$ NP $\cap$ co-NP. Încă o dată, nu se ştie dacă P = NP $\cap$ co-NP sau dacă există anumite limbaje în mulţimea NP $\cap$ co-NP $-$ P. Figura 36.2 arată cele 4 posibilităţi.

[4] Denumirea "NP" provine de la "nedeterminist polinomială". Clasa NP a fost studiată, la început, în contextul nedeterminismului, dar această carte foloseşte noţiunea, într-un anume fel, mai simplă, de verificare. Hopcroft şi Ullman [104] prezintă NP-completitudinea în termeni ai modelelor nedeterministe de calcul.

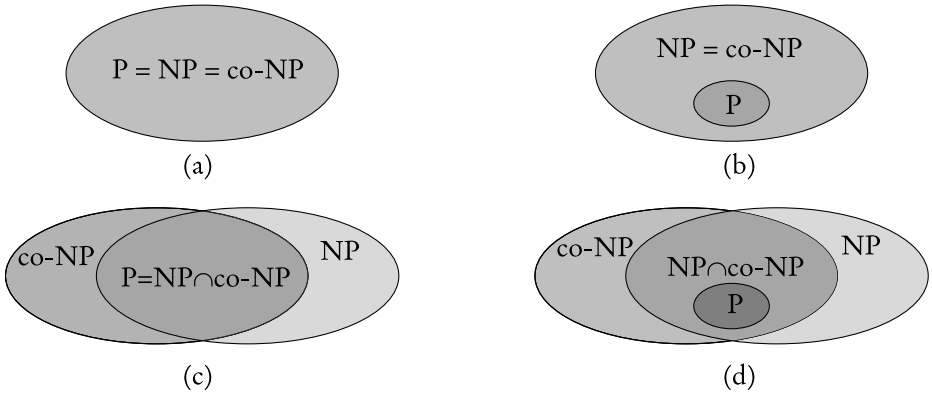


Figura 36.2 Patru posibile relaţii între clasele de complexitate. În fiecare diagramă, o regiune care include o altă regiune indică o relaţie de incluziune. **(a)** P = NP = co-NP. Cei mai mulţi cercetători privesc această posibilitate ca fiind cea mai puţin probabilă. **(b)** Dacă NP este închisă faţă de complement atunci NP = co-NP, dar aceasta nu înseamnă că P = NP. **(c)** P = NP ∩ co-NP, dar NP nu este închisă faţă de complement. **(d)** NP ≠ co-NP şi P ≠ NP ∩ co-NP. Cei mai mulţi cercetători privesc această posibilitate ca fiind cea mai probabilă.

Deci, cunoştinţele noastre asupra relaţiilor precise care există între P şi NP sunt lacunare. Cu toate acestea, cercetând teoria NP-completitudinii, vom afla că dezavantajul nostru în a demonstra că o problemă este inaccesibilă este, din punct de vedere practic, mult mai nesemnificativ decât am putea presupune.

Exerciţii

36.2-1 Să considerăm limbajul GRAF-IZO = $\{\langle G_1, G_2\rangle : G_1$ şi G_2 sunt grafuri izomorfe$\}$. Demonstraţi că GRAF-IZO ∈ NP, descriind un algoritm polinomial pentru verificarea limbajului.

36.2-2 Demonstraţi că, dacă G este un graf bipartit neorientat cu număr impar de vârfuri, G nu este hamiltonian.

36.2-3 Demonstraţi că, dacă CICLU-HAM ∈ P, atunci problema tipăririi în ordine a vârfurilor unui ciclu hamiltonian este rezolvabilă în timp polinomial.

36.2-4 Demonstraţi afirmaţia: clasa NP a limbajelor este închisă faţă de reuniune, intersecţie, concatenare şi steaua lui Kleene. Discutaţi închiderea clasei NP faţă de complement.

36.2-5 Demonstraţi că, orice limbaj din NP poate fi clarificat de un algoritm care se va executa într-un timp $2^{O(n^k)}$, unde k este o constantă.

36.2-6 Un ***drum hamiltonian*** într-un graf este un drum elementar care vizitează fiecare vârf exact o dată. Demonstraţi că limbajul DRUM-HAM = $\{\langle G, u, v\rangle$: există un drum hamiltonian de la u la v în graful $G\}$ aparţine clasei NP.

36.2-7 Arătaţi că problema drumului hamiltonian poate fi rezolvată în timp polinomial pentru grafuri orientate aciclice. Daţi un algoritm eficient pentru această problemă.

36.2-8 Fie o formulă logică ϕ construită cu ajutorul variabilelor logice de intrare $x_1, x_2, \ldots, x_k$, operatori de negaţie ($\neg$), ŞI-logic ($\wedge$), SAU-logic ($\vee$) şi paranteze. Formula ϕ este o ***tautologie*** dacă ia valoarea 1 pentru orice atribuire de 1 şi 0 dată variabilelor de intrare. Definiţi TAUTOLOGIE ca limbajul expresiilor logice care sunt tautologii. Arătaţi că TAUTOLOGIE $\in$ co-NP.

36.2-9 Arătaţi că P $\subseteq$ co-NP.

36.2-10 Arătaţi că dacă NP $\neq$ co-NP atunci P $\neq$ NP.

36.2-11 Fie G un graf neorientat conex cu cel puţin trei vârfuri şi fie G^3 graful obţinut prin unirea tuturor perechilor de vârfuri care sunt unite printr-un drum de lungime cel mult 3. Demonstraţi că graful G^3 este hamiltonian. (*Indicaţie:* construiţi un arbore parţial pentru G şi demonstraţi prin inducţie.)

36.3. NP-completitudine şi reductibilitate

Poate că cel mai atrăgător motiv, pentru care informaticienii cred că P $\neq$ NP, este existenţa clasei problemelor "NP-complete". Această clasă are proprietatea surprinzătoare că dacă o *singură* problemă NP-completă poate fi rezolvată în timp polinomial, atunci, pentru *toate* problemele NP-complete se va putea găsi un algoritm de rezolvare polinomial, adică P = NP. În ciuda anilor de studiu nu s-a descoperit nici un algoritm polinomial pentru rezolvarea vreunei probleme NP-complete.

Limbajul CICLU-HAM este o problemă NP-completă. Dacă am putea clarifica CICLU-HAM în timp polinomial, am putea rezolva orice problemă din NP în timp polinomial. De fapt, dacă s-ar dovedi că mulţimea NP $-$ P ar fi nevidă, atunci, am putea afirma cu certitudine că CICLU-HAM $\in$ NP $-$ P.

Limbajele NP-complete sunt, într-un anume sens, cele mai "dificile" limbaje din NP. În această secţiune, vom arăta cum putem compara "dificultatea" relativă a limbajelor folosind o noţiune numită "reductibilitate în timp polinomial". Mai întâi vom defini limbajele NP-complete, apoi vom demostra că un astfel de limbaj, numit CIRCUIT-SAT este o problemă NP-completă. În secţiunea 36.5 vom folosi noţiunea de reductibilitate pentru a arăta NP-completitudinea altor probleme.

Reductibilitate

Intuitiv, o problemă Q poate fi redusă la altă problemă Q', dacă orice instanţă a lui Q poate fi "uşor reformulată" ca o instanţă a problemei Q', iar găsirea unei soluţii pentru Q' va însemna găsirea unei soluţii pentru Q. De exemplu, problema rezolvării ecuaţiilor liniare cu o necunoscută x poate fi redusă la problema rezolvării ecuaţiilor pătratice cu o necunoscută. Dându-se o instanţă $ax + b = 0$, o transformăm în $0x^2 + ax + b = 0$, a cărei soluţie va fi de asemenea soluţie şi pentru $ax + b = 0$. Deci, dacă o problemă Q se reduce la o altă problemă Q', atunci Q nu este, într-un anume sens, "mai greu de rezolvat" decât Q'.

Revenind la limbajele formale pentru probleme de decizie, spunem că un limbaj L_1 este ***reductibil în timp polinomial*** la un limbaj L_2 dacă există o funcţie calculabilă în timp polinomial $f : \{0,1\}^* \to \{0,1\}^*$ astfel încât pentru orice $x \in \{0,1\}^*$,

$$x \in L_1 \text{ dacă şi numai dacă } f(x) \in L_2. \tag{36.1}$$

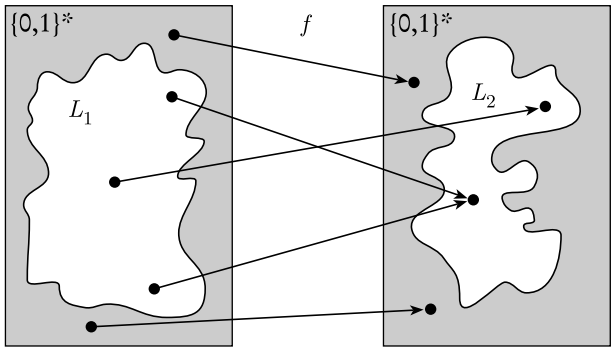


Figura 36.3 Ilustrarea reducerii în timp polinomial a unui limbaj L_1 la un limbaj L_2 prin intermediul funcţiei de reducere f. Pentru o intrare $x \in \{0,1\}^*$, întrebarea dacă $x \in L_1$ are acelaşi răspuns ca şi întrebarea dacă $f(x) \in L_2$.

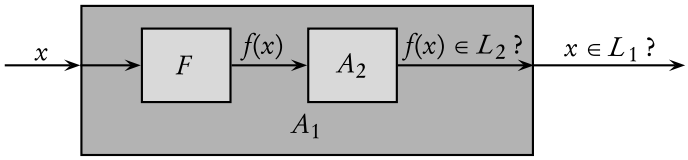


Figura 36.4 Demonstraţia lemei 36.3. Algoritmul F este un algoritm de reducere care calculează funcţia de reducere f de la L_1 la L_2 în timp polinomial şi A_2 este un algoritm polinomial care clarifică L_2. Este ilustrat un algoritm A_1 care decide dacă $x \in L_1$ folosind F pentru a transforma fiecare intrare x în $f(x)$ şi folosind apoi A_2 pentru a decide dacă $f(x) \in L_2$.

Vom folosi notaţia $L_1 \leq_{\mathrm{P}} L_2$ pentru a arăta că limbajul L_1 este reductibil în timp polinomial la limbajul L_2. Funcţia f se numeşte ***funcţie de reducere*** şi algoritmul polinomial F care calculează f se numeşte ***algoritm de reducere***.

În figura 36.3 este ilustrată ideea unei reduceri în timp polinomial de la un limbaj L_1 la un limbaj L_2. Fiecare limbaj este o submulţime a mulţimii $\{0,1\}^*$. Funcţia de reducere f furnizează, în timp polinomial, o corespondenţă între x şi $f(x)$ astfel încât $x \in L_1$ şi $f(x) \in L_2$. Mai mult dacă $x \notin L_1$ atunci $f(x) \notin L_2$. Deci funcţia de reducere realizează o corespondenţă între orice instanţă x a problemei de decizie reprezentată de limbajul L_1 şi o instanţă $f(x)$ a problemei reprezentate de L_2. Obţinând un răspuns pentru întrebarea $f(x) \in L_2$ avem răspuns şi pentru întrebarea $x \in L_1$.

Reducerile în timp polinomial ne oferă o unealtă puternică pentru a demonstra că diferite limbaje fac parte din P.

Lema 36.3 Dacă $L_1, L_2 \subseteq \{0,1\}^*$ sunt limbaje astfel încât $L_1 \leq_{\mathrm{P}} L_2$, atunci, $L_2 \in \mathrm{P}$ implică $L_1 \in \mathrm{P}$.

Demonstraţie. Fie A_2 un algoritm polinomial care clarifică L_2 şi F un algoritm polinomial de reducere care calculează funcţia de reducere f. Vom construi un algoritm polinomial A_1 care clarifică L_1.

În figura 36.4 este ilustrat modul în care este construit A_1. Pentru o intrare dată $x \in \{0,1\}^*$,

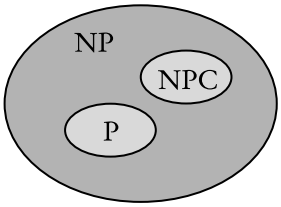


Figura 36.5 Modul în care sunt privite, de majoritatea informaticienilor, relaţiile dintre P, NP şi NPC. Atât P cât şi NPC sunt incluse în NP şi P $\cap$ NPC $= \emptyset$.

algoritmul A_1 foloseşte F pentru a transforma x în $f(x)$ şi apoi foloseşte A_2 pentru a testa dacă $f(x) \in L_2$. Intrarea lui A_2 este valoarea furnizată ca ieşire de A_1.

Corectitudinea lui A_1 rezultă din condiţia (36.1). Algoritmul se execută în timp polinomial deoarece atât F cât şi A_2 se execută în timp polinomial (vezi exerciţiul 36.1-6). ■

NP-completitudine

Reducerile în timp polinomial furnizează o modalitate pentru a arăta că o problemă este cel puţin la fel de dificilă ca alta, apărând, în plus sau în minus, cel mult un factor polinomial. Aceasta înseamnă că dacă $L_1 \leq_{\mathrm{P}} L_2$ atunci L_1 este cu cel mult un factor polinomial mai dificilă decât L_2. De aici derivă şi semnul de "mai mic sau egal" din notaţia pentru reducere. Putem defini acum mulţimea limbajelor NP-complete, mulţime care conţine cele mai dificile probleme din NP.

Un limbaj $L \subseteq \{0,1\}^*$ este ***NP-complet*** dacă

1. $L \in$ NP, şi
2. $L' \leq_{\mathrm{P}} L$, oricare ar fi $L' \in$ NP.

Dacă un limbaj L satisface proprietatea 2, dar nu neapărat şi propritatea 1, atunci spunem că acest limbaj este ***NP-dificil***. Definim, de asemenea, clasa NPC ca fiind clasa limbajelor NP-complete.

După cum arată teorema următoare, NP-completitudinea este un element cheie în a decide că P este, de fapt, egală cu NP.

Teorema 36.4 Dacă orice problemă NP-completă este rezolvabilă în timp polinomial, atunci P = NP. De asemenea, dacă orice problemă din NP nu este rezolvabilă în timp polinomial, problemele NP-complete nu sunt rezolvabile în timp polinomial.

Demonstraţie. Să presupunem că $L \in$ P şi $L \in$ NPC. Din proprietatea 2, care apare în definiţia NP-completitudinii rezultă că pentru orice $L' \in$ NP, avem $L' \leq_{\mathrm{P}} L$. Deci, conform lemei 36.3, rezultă că $L' \in$ P, ceea ce demonstrează prima parte a teoremei.

Pentru a demonstra a doua parte a teoremei, observaţi că este reciproca primei părţi. ■

Din această cauză, cercetarea problemei P $\neq$ NP se axează în jurul problemelor NP-complete. Cei mai mulţi informaticieni cred că P $\neq$ NP, ceea ce ar duce la relaţiile dintre P, NP şi NPC ilustrate în figura 36.5. Dar, din câte ştim, cineva ar putea găsi un algoritm polinomial pentru o problemă NP-completă demonstrând astfel că P = NP. Cu toate acestea, datorită faptului că nu

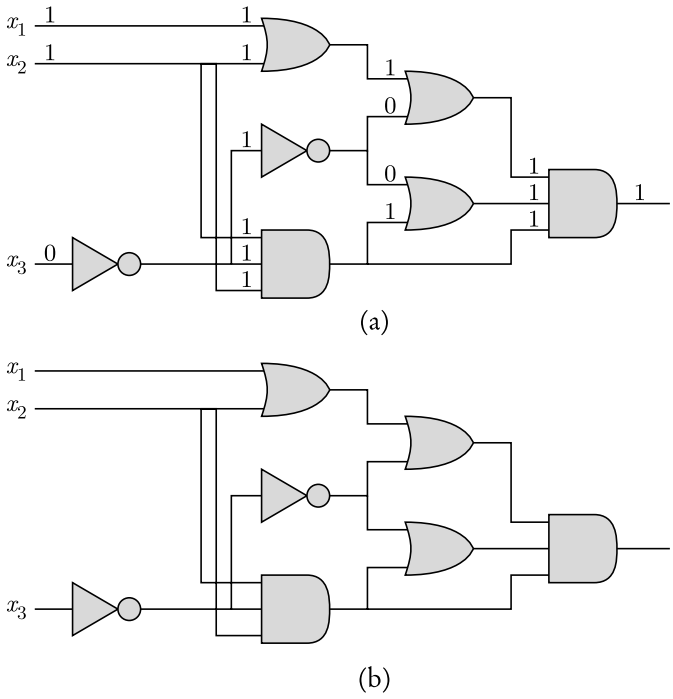


Figura 36.6 Două instanţe ale problemei satisfiabilităţii circuitului. **(a)** Valorile $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ pentru intrările circuitului determină ieşirea 1, deci circuitul este satisfiabil. **(b)** Indiferent de valorile pe care le iau intrările circuitului, ieşirea va fi întotdeauna 1, deci circuitul nu este satisfiabil.

s-a găsit încă nici un algoritm polinomial pentru vreo problemă NP-completă, o demonstraţie a faptului că o problemă este NP-completă este acceptată ca o dovadă excelentă a inaccesibilităţii acestei probleme.

Satisfiabilitatea circuitului

Am definit noţiunea de problemă NP-completă, dar, până acum, nu am demonstrat că o anumită problemă este NP-completă. După ce vom dovedi că cel puţin o problemă este NP-completă, vom putea folosi reductibilitatea polinomială pentru a dovedi NP-completitudinea altor probleme. Ne vom concentra acum atenţia asupra demonstrării existenţei unei probleme NP-complete: problema satisfiabilităţii circuitului.

Din nefericire, demonstraţia riguroasă a faptului că problema satisfiabilităţii circuitului este NP-completă necesită detalii tehnice care depăşesc scopul lucrării de faţă. În locul unei astfel de demonstraţii, vă vom prezenta o demonstraţie care se bazează pe noţiuni fundamentale despre circuitele combinaţionale logice. Acest material este recapitulat la începutul capitolului 29.

Figura 36.6 prezintă două circuite combinaţionale logice, fiecare având trei intrări şi o singură ieşire. O ***atribuire de adevăr*** a unui circuit este o mulţime de valori logice de intrare. Spunem că un circuit cu o singură ieşire este ***satisfiabil*** dacă are o ***atribuire satisfiabilă***: o atribuire de adevăr care determină ieşirea 1. De exemplu, circuitul din figura 36.6(a) are atribuirea satisfiabilă

$\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$, deci este satisfiabil. Nici o atribuire pentru valorile x_1, x_2 şi x_3 nu poate determina ieşirea 1, aceasta fiind întotdeauna 0, deci circuitul nu este satisfiabil.

Problema satisfiabilităţii circuitului poate fi enunţată astfel: "Dându-se un circuit combinaţional logic compus din porţi ŞI, SAU şi NU, este el satisfiabil?" Pentru a putea formula această întrebare, trebuie să cădem de acord asupra unei codificări standard pentru circuite. Putem accepta o codificare de tipul grafurilor care stabileşte o corespondenţă între un circuit C şi un şir binar $\langle C\rangle$ a cărui lungime nu este mult mai mare decât mărimea circuitului. Putem defini limbajul formal:

$$\textsc{Circuit-Sat} = \{\langle C\rangle : C\text{este un circuit combinaţional logic care este satisfiabil}\}.$$

Problema satisfiabilităţii circuitului are o mare importanţă în domeniul optimizărilor hardware asistate de calculator. Dacă ieşirea circuitului este întotdeauna 0, acesta poate fi înlocuit cu un circuit simplu în care sunt eliminate toate porţile logice şi are ca ieşire valoarea constantă 0. Un algoritm polinomial pentru această problemă ar avea o deosebită aplicabilitate practică.

Dându-se un circuit C, am putea testa dacă acesta este satisfiabil încercând toate posibilităţile pentru intrări. Din nefericire, pentru k intrări am avea 2^k combinaţii posibile. Dacă mărimea circuitului este dată de o funcţie polinomială care depinde de k, verificarea tuturor posibilităţilor duce la un algoritm suprapolinomial. De fapt, aşa cum am afirmat anterior, există o demonstraţie riguroasă a faptului că nu există nici un algoritm polinomial care rezolvă problema satisfiabilităţii circuitului deoarece această problemă este NP-completă. Vom împărţi demonstraţia acestui fapt în două părţi, bazate pe cele două părţi ale definiţiei NP-completitudinii.

Lema 36.5 Problema satisfiabilităţii circuitului aparţine clasei NP.

Demonstraţie. Vom descrie un algoritm polinomial A cu două intrări care verifică Circuit-Sat. Una din intrările pentru A este reprezentată de o codificare standard a unui circuit combinaţional logic C. Cealaltă intrare este o probă corespunzătoare unei atribuiri de valori logice pentru firele din C.

Algoritmul A este construit după cum urmează. Pentru fiecare poartă logică a circuitului, verifică dacă valoarea furnizată de probă la firul de ieşire este corect calculată ca o funcţie de valorile de la firele de intrare. Atunci, dacă ieşirea întregului circuit este 1, algoritmul va avea ca ieşire valoarea 1 deoarece valorile atribuite intrărilor lui C furnizează o atribuire satisfiabiliă. În caz contrar, programul va avea ca ieşire valoarea 0.

De fiecare dată, când un circuit satisfiabil C este intrare pentru algoritmul A, există o probă a cărei lungime este o funcţie polinomială care depinde de mărimea lui C şi care determină algoritmul A să furnizeze ieşirea 1. De fiecare dată, când un circuit care nu este satisfiabil este intrare pentru algoritmul A, nici o probă nu va putea păcăli algoritmul A. Acest algoritm se execută în timp polinomial: dacă este bine implementat, este suficient un timp liniar de execuţie. Deci, Circuit-Sat poate fi verificat în timp polinomial, ca urmare Circuit-Sat $\in$ NP. ∎

Pentru a doua parte a demonstraţiei faptului că limbajul Circuit-Sat este NP-complet, trebuie să arătăm că acest limbaj este NP-dificil. Aceasta înseamnă că va trebui să arătăm că orice limbaj din NP este reductibil în timp polinomial la Circuit-Sat. Demonstraţia riguroasă a acestui fapt conţine multe complicaţii tehnice, de aceea nu o vom reproduce aici. Vom schiţa, totuşi, o demonstraţie bazată pe cunoaşterea unor detalii ale modului în care funcţionează partea hardware a calculatorului.

Un program este stocat în memoria calculatorului ca o secvenţă de instrucţiuni. O instrucţiune obişnuită va codifica o operaţie care trebuie executată, adresele operanzilor din memorie şi adresa la care trebuie stocat rezultatul. O locaţie specială de memorie, numită ***registru instrucţiune program***, păstrează adresa instrucţiunii care trebuie executată la pasul următor. Valoarea acestui registru este incrementată automat ori de câte ori se încheie execuţia unei instrucţiuni, ceea ce duce la o execuţie secvenţială a instrucţiunilor. Execuţia unei instrucţiuni poate duce la scrierea unei anumite valori în acest registru, astfel că execuţia instrucţiunilor nu va mai fi secvenţială, permiţând calculatorului să execute cicluri sau instrucţiuni de tipul alternative.

În orice moment al execuţiei programului, starea tuturor factorilor care influenţează calculele este păstrată în memoria calculatorului. (Considerăm că memoria include programul, registrul instrucţiune program, memoria de lucru şi diferiţi biţi de stare pe care îi păstrează calculatorul pentru contabilizare.) Numim ***configuraţie*** o anumită stare a memoriei calculatorului. Execuţia unei instrucţiuni poate fi văzută ca o funcţie între două configuraţii. Important este faptul că partea hardware a calculatorului care realizează această corespondenţă poate fi implementată ca un circuit combinaţional logic pe care îl vom nota cu M în demonstraţia lemei următoare.

Lema 36.6 Problema satisfiabilităţii circuitului este NP-dificilă.

Demonstraţie. Fie L un limbaj din NP. Vom descrie un algoritm polinomial F pentru calculul funcţiei f de reducere care realizează o corespondenţă între un şir binar x şi un circuit $C = f(x)$, astfel încât $x \in L$ dacă, şi numai dacă, $C \in$ CIRCUIT-SAT.

Cum $L \in$ NP, trebuie să existe un algoritm A care verifică L în timp polinomial. Algoritmul F pe care îl vom construi va folosi algoritmul cu două intrări A pentru a calcula funcţia de reducere f.

Notăm cu $T(n)$ timpul de rulare a algoritmului A în cazul cel mai defavorabil când acesta se execută cu şiruri de intrare de lungime n. Fie $k \geq 1$ o constantă astfel încât $T(n) = O(n^k)$ şi lungimea probei este $O(n^k)$. (Timpul de execuţie al algoritmului A depinde după o funcţie polinomială de lungimea totală a intrării, ce conţine atât şirul de intrare cât şi proba. Deoarece lungimea probei depinde după o funcţie polinomială de lungimea n a şirului de intrare, timpul de execuţie va fi o funcţie polinomială în n.)

Ideea de bază a demonstraţiei este să reprezentăm calculele efectuate de A ca o serie de configuraţii. După cum se arată în figura 36.7, fiecare configuraţie poate fi împărţită în mai multe părţi: programul care implementează algoritmul A, registrul instrucţiune program şi starea maşinii, intrarea x, proba y şi memoria de lucru. Pornind de la o configuraţie iniţială c_0, fiecare configuraţie c_i este pusă în corespondenţă cu configuraţia următoare c_{i+1} de către circuitul combinaţional M care implementează partea hardware a calculatorului. Ieşirea algoritmului A – 0 sau 1 – este scrisă, la încheierea execuţiei programului care îl implementează, într-o locaţie a spaţiului de lucru special aleasă în acest scop. Dacă presupunem că programul care implementează A se opreşte, atunci această valoare nu va mai fi modificată. Deci algoritmul va avea cel mult $T(n)$ paşi, şi ieşirea va fi unul dintre biţii din $c_{T(n)}$.

Algoritmul de reducere F construieşte un singur circuit combinaţional care calculează toate configuraţiile rezultate din configuraţia iniţială. Ideea este de a uni $T(n)$ copii ale circuitului M. Ieşirea celui de-al i-lea circuit, care determină configuraţia c_i este conectată la intrarea celui de-al i+1-lea circuit. Deci, configuraţiile, în loc să fie păstrate în regiştrii de stare, vor consta în valori pe firele copiilor conectate ale lui M.

Să ne amintim ce trebuie să realizeze algoritmul polinomial de reducere F. Dându-se o intrare x, el trebuie să calculeze un circuit $C = f(x)$ care este satisfiabil dacă, şi numai dacă, există o

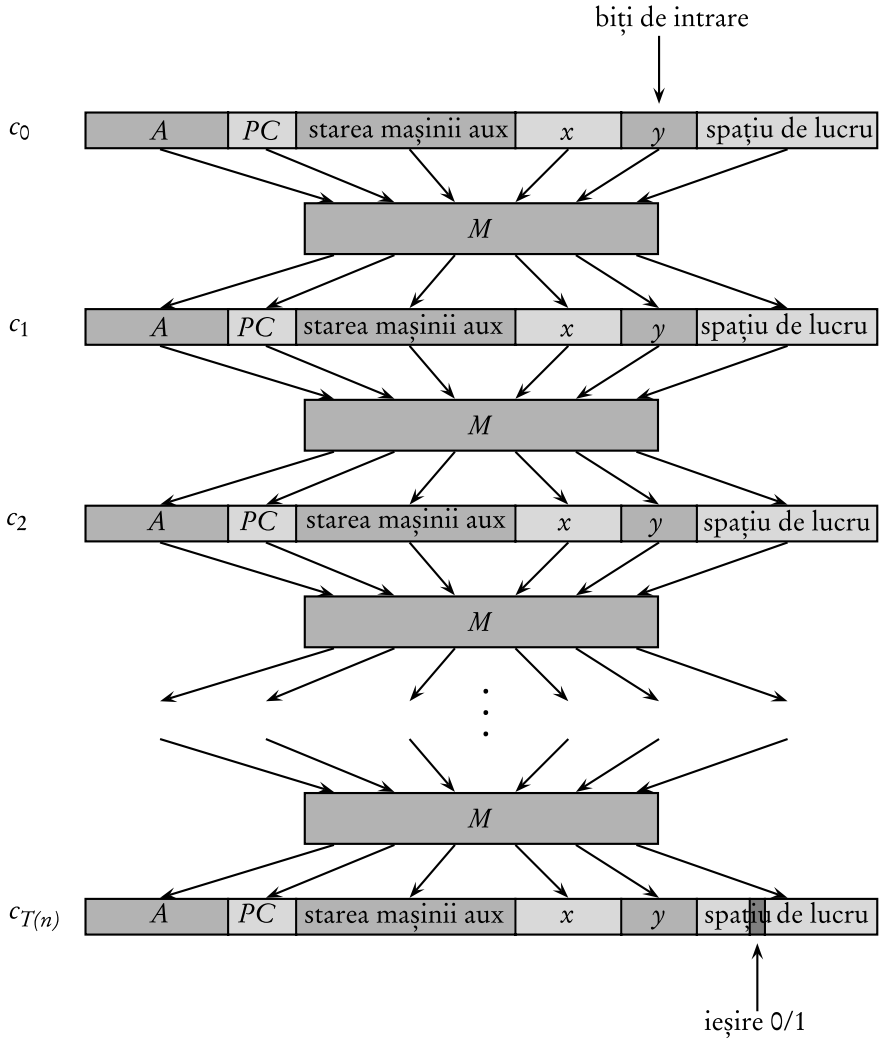


Figura 36.7 Seria configuraţiilor determinate de algoritmul A care se execută pentru o intrare x şi o probă y. Fiecare configuraţie reprezintă starea calculatorului pentru un anumit pas şi include, pe lângă A, x şi y, registrul instrucţiune program, starea maşinii şi memoria de lucru. Cu excepţia probei y, configuraţia iniţială c_0 este constantă. Fiecare configuraţie este pusă în corespondenţă ca următoarea printr-un circuit combinaţional logic M. Ieşirea este un anumit bit din memoria de lucru.

probă y, astfel încât $A(x, y) = 1$. Când F obţine o intrare x, el calculează mai întâi $n = |x|$ şi construieşte un circuit combinaţional C' care constă din $T(n)$ copii ale lui M. Intrarea lui C' este o configuraţie iniţială corespunzătoare unui calcul pe $A(x, y)$, iar ieşirea este configuraţia $c_{T(n)}$.

Circuitul $C = f(x)$ pe care îl calculează F este obţinut făcând o mică modificare la circuitul C'. Mai întâi, intrările pentru C' corespunzătoare programului care implementează A, valoarea

iniţială a registrului instrucţiune program, intrarea x şi starea iniţială a memoriei sunt puse în legătură cu aceste valori cunoscute. Deci, singurele intrări rămase ale circuitului sunt cele corespunzătoare probei y. Apoi toate ieşirile circuitului sunt ignorate cu excepţia singurului bit din $c_{T(n)}$ care corespunde ieşirii lui A. Circuitul C, construit în acest mod, calculează $C(y) = A(x, y)$ pentru orice intrare y de lungime $O(n^k)$. Algoritmul de reducere F rezolvă un astfel de circuit C când primeşte la intrare şirul x.

Mai rămân de demonstrat două proprietăţi. În primul rând trebuie să arătăm că F calculează în mod corect o funcţie de reducere f, respectiv trebuie să arătăm că C este satisfiabil dacă, şi numai dacă, există o probă y astfel încât $A(x, y) = 1$. În al doilea rând, trebuie demonstrat şi faptul că F se execută în timp polinomial.

Pentru a arăta că F calculează în mod corect o funcţie de reducere, să presupunem că există o probă y de lungime $O(n^k)$ astfel încât $A(x, y) = 1$. Atunci, dacă furnizăm biţii lui y ca intrări pentru C, ieşirea lui C va fi $C(y) = A(x, y) = 1$. Deci, dacă există o probă, C este satisfiabil. Acum, să presupunem că C este satisfiabil. Ca urmare există o intrare y pentru C astfel încât, $C(y) = 1$. De aici tragem concluzia că $A(x, y) = 1$, deci F calculează corect o funcţie de reducere.

Pentru a încheia demonstraţia, mai trebuie, doar, să arătăm că F se execută în timp polinomial pentru $n = |x|$. Prima observaţie pe care o facem este că numărul de biţi necesari pentru a reprezenta o configuraţie depinde după o funcţie polinomială de n. Programul care implementează A are o mărime constantă, independentă de lungimea intrării x. Lungimea intrării x este n, iar lungimea probei y este $O(n^k)$. Deoarece algoritmul va avea cel mult $O(n^k)$ paşi, spaţiul necesar pentru memorarea lui A depinde, de asemenea, după o funcţie polinomială de n. (Presupunem că memoria este contiguă; exerciţiul 36.3-4 vă cere să extindeţi demonstraţia pentru cazul în care locaţiile accesate de A sunt răspândite pe o zonă mai mare de memorie şi şablonul răspândirii poate să difere pentru fiecare intrare x.)

Circuitul combinaţional M care implementează partea hardware a calculatorului are dimensiunea dependentă după o funcţie polinomială de lungimea unei configuraţii, lungime care este polinomială în $O(n^k)$, deci este polinomială în n. (Cea mai mare parte a acestei scheme implementează partea logică a memoriei sistem.) Circuitul C constă în cel mult $t = O(n^k)$ copii ale lui M, deci dimensiunea lui depinde de o funcţie polinomială în n. Construirea lui C din x poate fi realizată în timp polinomial de către algoritmul de reducere F, deoarece fiecare pas necesită un timp polinomial. ∎

În concluzie, limbajul CIRCUIT-SAT este cel puţin la fel de dificil ca orice alt limbaj din NP şi, datorită faptului că aparţine clasei NP, este NP-complet.

Teorema 36.7 Problema satisfiabilităţii circuitului este NP-completă.

Demonstraţie. Rezultă imediat din lemele 36.5 şi 36.6 şi definiţia NP-completitudinii. ∎

Exerciţii

36.3-1 Arătaţi că relaţia $\leq_P$ este o relaţie tranzitivă între limbaje, respectiv dacă $L_1 \leq_P L_2$ şi $L_2 \leq_P L_3$ atunci $L_1 \leq_P L_3$.

36.3-2 Arătaţi că $L \leq_P \overline{L}$ dacă, şi numai dacă, $\overline{L} \leq_P L$.

36.3-3 Arătaţi că o atribuire satisfiabilă poate fi folosită ca probă într-o demonstraţie alternativă a lemei 36.5. Care probă credeţi că uşurează demonstraţia?

36.3-4 În demonstraţia lemei 36.6 s-a presupus că memoria de lucru pentru algoritmul A este o zonă contiguă de memorie cu o dimensiune polinomială. Unde este folosită această presupunere în demonstraţie? Demonstraţi că această presupunere nu implică restrângerea generalităţii.

36.3-5 Un limbaj L este ***complet*** pentru o clasă de limbaje C cu privire la reducerile în timp polinomial dacă $L \in C$ şi $L' \leq_P L$, oricare ar fi $L' \in C$. Arătaţi că $\emptyset$ şi $\{0,1\}^*$ sunt singurele limbaje din P care nu sunt complete pentru P cu privire la reducerile în timp polinomial.

36.3-6 Arătaţi că L este complet pentru NP dacă şi numai dacă $\overline{L}$ este complet pentru co-NP.

36.3-7 Algoritmul de reducere F folosit în demonstraţia lemei 36.6, construieşte circuitul $C = f(x)$ bazat pe cunoaşterea lui x, A şi k. Profesorul Sartre a observat că şirul x este intrare pentru F, dar numai existenţa lui A şi k este cunoscută de F (deoarece limbajul L aparţine clasei NP), nu şi valorile lor. Profesorul a concluzionat că F nu poate construi circuitul C şi că limbajul CIRCUIT-SAT nu este neapărat NP-dificil. Găsiţi şi explicaţi greşeala din raţionamentul profesorului.

36.4. Demonstraţii ale NP-completitudinii

NP-completitudinea problemei satisfiabilităţii circuitului se bazează pe dovada directă că $L \leq_P$ CIRCUIT-SAT pentru toate limbajele $L \in$ NP. În această secţiune, vom arăta cum se poate demonstra că un limbaj este NP-complet fără a reduce, în mod direct, *fiecare* limbaj din NP la acel limbaj. Vom ilustra această metodologie demonstrând NP-completitudinea mai multor probleme de satisfiabilitate a unei anumite formule. În secţiunea 36.5 sunt prezentate mai multe exemple ale acestei metodologii.

Următoarea lemă reprezintă baza de la care porneşte metoda noastră de a demonstra că un anumit limbaj este NP-complet.

Lema 36.8 Dacă L este un limbaj astfel încât $L' \leq_P L$ pentru $L' \in$ NPC, atunci L este NP-dificil. Mai mult, dacă $L \in$ NP, atunci, $L \in$ NPC.

Demonstraţie. Deoarece L' este NP-complet, pentru toate limbajele $L'' \in$ NP avem $L'' \leq_P L'$. Din ipoteza $L' \leq_P L$ şi datorită proprietăţii de tranzitivitate a relaţiei "$\leq_P$" (exerciţiul 36.3-1) obţinem că $L'' \leq_P L$, ceea ce arată că L este NP-dificil. Dacă $L \in NP$, avem, de asemenea, $L \in$ NPC. ∎

Cu alte cuvinte, reducând la L un limbaj NP-complet L', reducem la L, în mod implicit, orice limbaj din NP. Deci, lema 36.8 ne oferă o metodă de a demonstra că un limbaj L este NP-complet:

1. Demonstraţi că $L \in$ NP.
2. Alegeţi un limbaj L' despre care ştiţi că este NP-complet
3. Descrieţi un algoritm care calculează o funcţie f care pune în corespondenţă fiecare instanţă a lui L' cu o instanţă a lui L.

4. Demonstraţi că funcţia f satisface condiţia $x \in L'$ dacă, şi numai dacă, $f(x) \in L$ pentru toţi $x \in \{0,1\}^*$.

5. Demonstraţi că algoritmul care calculează f se execută în timp polinomial.

Această metodologie care foloseşte reducerea unui singur limbaj NP-complet este mult mai uşor de folosit decât procedura complicată de a reduce toate limbajele din NP. Demonstrând că CIRCUIT-SAT $\in$ NPC "am pus un picior în pragul uşii" pentru că, ştiind că problema satisfiabilităţii circuitului este NP-completă, putem demonstra, mult mai uşor, că alte probleme sunt NP-complete. Mai mult, cu cât vom găsi mai multe probleme NP-complete, folosirea acestei metodologii va deveni mult mai uşoară.

Satisfiabilitatea formulelor

Ilustrăm metodologia reducerii dând o demonstraţie a NP-completitudinii problemei care verifică dacă o formulă logică, nu un circuit, este satisfiabilă. Această problemă are onoarea istorică de a fi prima problemă pentru care a existat o demonstraţie a NP-completitudinii.

Formulăm problema ***satisfiabilităţii (formulei)*** în termenii limbajului SAT astfel: instanţă SAT este o formulă booleană ϕ compusă din:

1. variabile logice: $x_1, x_2, \ldots$;

2. legături logice: orice funcţie booleană cu una sau două intrări şi o singură ieşire cum ar fi $\wedge$(ŞI), $\vee$(SAU), $\neg$(NU), $\rightarrow$(implicaţie), $\leftrightarrow$(dacă şi numai dacă);

3. paranteze.

Asemănător circuitelor combinaţionale, o ***atribuire de adevăr***, pentru o formulă booleană ϕ, este o mulţime de valori pentru variabilele lui ϕ, iar o ***atribuire satisfiabilă*** este o atribuire de adevăr pentru care evaluarea expresiei duce la rezultatul 1. O formulă care are o atribuire satisfiabilă este o formulă ***satisfiabilă***. Problema satisfiabilităţii cere determinarea faptului dacă o anumită formulă este sau nu satisfiabilă; în termenii limbajelor formale:

SAT $= \{\langle\phi\rangle : \phi$ este o formulă logică satisfiabilă$\}$.

De exemplu formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

are atribuirea satisfiabilă $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, deoarece

$$\phi = ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 = (1 \vee \neg(1 \vee 1)) \wedge 1 = (1 \vee 0) \wedge 1 = 1, \tag{36.2}$$

deci formula ϕ aparţine lui SAT.

Algoritmul naiv de determinare a faptului dacă o formulăeste sau nu satisfiabilă nu se execută în timp polinomial. Pot exista 2^n atribuiri de adevăr posibile pentru o furmulă ϕ cu n variabile. Dacă lungimea lui $\langle\phi\rangle$ este polinomială în n, atunci verificarea fiecărei atribuiri de adevăr necesită un timp suprapolinomial. După cum arată şi teorema următoare este puţin probabil să existe un algoritm polinomial.

Teorema 36.9 Problema satisfiabilităţii formulei logice este NP-completă.

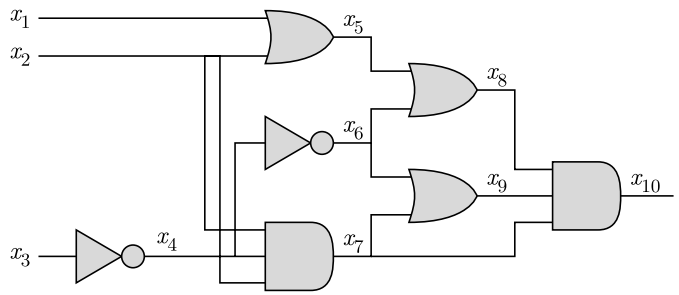


Figura 36.8 Reducerea problemei satisfiabilităţii circuitului la problema satisfiabilităţii formulei. Formula rezultată în urma algoritmului de reducere are câte o variabilă pentru fiecare fir din circuit.

Demonstraţie. Vom demonstra, mai întâi, că SAT $\in$ NP şi apoi, vom arăta că CIRCUIT-SAT $\leq_P$ SAT; potrivit lemei 36.8, aceste două demonstraţii sunt suficiente pentru a demonstra teorema.

Pentru a demonstra că SAT $\in$ NP, arătăm că o probă, care constă într-o atribuire satisfiabilă pentru o formulă ϕ dată la intrare, poate fi verificată în timp polinomial. Algoritmul de verificare va înlocui fiecare variabilă din formulă cu valoarea corespunzătoare şi va evalua expresia într-o manieră asemănătoare celei folosite pentru evaluarea expresiei (36.2). Această sarcină poate fi îndeplinită foarte uşor în timp polinomial. Dacă valoarea rezultată în urma evaluării va fi 1 atunci formula este satisfiabilă. Deci, prima condiţie din lema 36.8 este îndeplinită.

Pentru a demonstra că SAT este NP-dificilă, arătăm că CIRCUIT-SAT $\leq_P$ SAT. Cu alte cuvinte, orice instanţă a problemei satisfiabilităţii circuitului poate fi redusă în timp polinomial la o instanţă a problemei satisfiabilităţii formulei. Prin inducţie, putem arăta că orice circuit combinaţional logic poate fi exprimat ca o formulă logică. Vom lua în considerare poarta a cărei ieşire reprezintă şi ieşirea circuitului şi apoi, prin inducţie, exprimăm intrările porţii ca formule. Formula circuitului este obţinută scriind o expresie care aplică funcţia porţii la formulele intrărilor.

Din nefericire, această metodă directă nu constituie o reducere în timp polinomial, subformulele distribuite ar putea duce la creşterea exponenţială a formulei generate (vezi exerciţiul 36.4-1). Deci, algoritmul de reducere trebuie să fie, într-un anume fel, mai performant.

Figura 36.8 ilustrează ideea de bază a reducerii de la CIRCUIT-SAT la SAT pentru circuitul din figura 36.6(a). Pentru fiecare fir x_i al circuitului C, formula ϕ are o variabilă x_i. Operaţia unei porţi poate fi acum exprimată ca o formulă care implică variabilele firelor incidente. De exemplu, operaţia porţii de ieşire ŞI este $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$.

Formula ϕ, rezultată în urma algoritmului de reducere, este un ŞI între variabila de ieşire a circuitului şi conjuncţia propoziţiilor care descriu operaţiile fiecărei porţi. Pentru circuitul din figură, formula este:

$$\begin{aligned}\phi = {} & x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \wedge (x_6 \leftrightarrow \neg x_4) \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)).\end{aligned}$$

Dându-se un circuit C, este uşor să obţinem o astfel de formulă ϕ în timp polinomial.

De ce este circuitul C satisfiabil când formula ϕ este satisfiabilă? Dacă C are o atribuire satisfiabilă, atunci fiecare fir al circuitului are o valoare bine definită şi ieşirea circuitului este 1. De aceea, atribuirea valorilor firelor pentru variabilele din ϕ duce la evaluarea la 1 a fiecărei propoziţii din ϕ, deci conjuncţia tuturor va fi evaluată la 1. Analog, dacă există o atribuire de

adevăr care duce la evaluarea la 1 a expresiei, circuitul C este satisfiabil. Am arătat, deci, că CIRCUIT-SAT $\leq_P$ SAT, ceea ce încheie demonstraţia. ∎

Problema satisfiabilităţii 3-FNC

NP-completitudinea multor probleme poate fi demonstrată prin reducerea problemei satisfiabilităţii formulei logice la aceste probleme. Algoritmul de reducere trebuie să manipuleze orice formulă dată la intrare, cu toate că aceasta ar putea duce la un număr uriaş de cazuri care trebuie luate în considerare. De aceea, adesea este de dorit să facem o reducere de la un limbaj restrâns pentru formulele logice, pentru ca numărul cazurilor care trebuie luate în considerare să scadă. Bineînţeles, nu avem voie să restrângem limbajul atât de mult, încât să devină rezolvabil în timp polinomial. Un limbaj convenabil este satisfiabilitatea 3-FNC, pe care îl vom nota cu 3-FNC-SAT.

Definim problema satisfiabilităţii 3-FNC folosind următorii termeni: un ***literal*** într-o formulă logică este o apariţie a unei variabile sau a negaţiei acesteia; o formulă logică se află în ***formă normal-conjuctivă***, sau ***FNC***, dacă este exprimată ca fiind conjuncţia mai multor propoziţii, fiecare propoziţie fiind formată din disjuncţia mai multor literali. O formulă logică se află în ***a treia formă normal-conjuctivă***, sau ***3-FNC***, dacă fiecare propoziţie conţine exact trei literali distincţi.

De exemplu, formula logică

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

este în a treia formă normal-conjuctivă. Prima din cele trei propoziţii este $(x_1 \vee \neg x_1 \vee \neg x_2)$ şi conţine trei literali: x_1, $\neg x_1$ şi $\neg x_2$.

În 3-FNC-SAT ni se cere să determinăm dacă o formulă logică ϕ în 3-FNC este satisfiabilă. Următoarea teoremă arată că este puţin probabil să existe un algoritm polinomial care verifică dacă o formulă logică este satisfiabilă, chiar dacă este exprimată în această formă simplă.

Teorema 36.10 Problema satisfiabilităţii formulelor logice în a treia formă normal-conjunctivă este NP-completă.

Demonstraţie. Argumentele pe care le-am folosit în demonstrarea teoremei 36.9, pentru a arăta că SAT $\in$ NP, pot fi folosite şi pentru a demonstra că 3-FNC-SAT $\in$ NP. Deci mai trebuie să arătăm numai că 3-FNC-SAT este NP-dificilă. Vom demonstra acest fapt arătând că SAT $\leq_P$ 3-FNC-SAT şi, de aici, va rezulta, conform lemei 36.8, că 3-FNC-SAT este NP-completă.

Algoritmul de reducere poate fi divizat în trei paşi elementari. La fiecare pas vom scrie formula ϕ într-o formă tot mai apropiată de 3-FNC.

Primul pas este similar cu demonstraţia faptului că CIRCUIT-SAT $\leq_P$ SAT, din teorema 36.9. Mai întâi, construim un arbore "gramatical" pentru formula ϕ, având literalii ca frunze şi legăturile ca noduri interne. Figura 36.9 arată un astfel de arbore gramatical pentru formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2. \tag{36.3}$$

În cazul în care formula conţine o propoziţie de tip SAU între mai mulţi literali, proprietatea de asociativitate poate fi folosită pentru a paranteza toată expresia în aşa fel încât fiecare nod intern al arborelui rezultat să aibă unul sau doi fii. Arborele gramatical binar poate fi acum privit ca un circuit pentru calculul formulei logice.

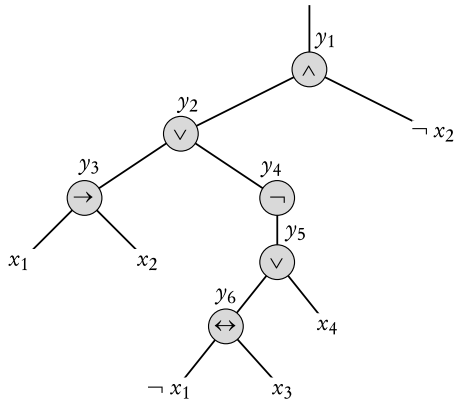


Figura 36.9 Arborele corespunzător formulei $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

Imitând reducerea din demonstraţia teoremei 36.9, introducem o variabilă y_i pentru ieşirea fiecărui nod intern. Apoi, rescriem formula iniţială ϕ ca un ŞI între variabila din rădăcină şi o conjucţie a propoziţiilor care descriu operaţiile din fiecare nod. Pentru formula (36.3) expresia rezultată este:

$$\begin{aligned}\phi' = y_1 \quad &\wedge \quad (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ &\wedge \quad (y_4 \leftrightarrow \neg y_5) \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)).\end{aligned}$$

Se observă că formula ϕ' obţinută este o conjucţie de propoziţii ϕ'_i în formă normal-conjuctivă, fiecare propoziţie având cel mult trei literali. Singura condiţie care mai trebuie pusă este ca fiecare propoziţie să fie o disjuncţie între literali.

La al doilea pas al reducerii, vom converti fiecare propoziţie ϕ'_i în formă normal-conjunctivă. Vom construi o tabelă de adevăr pentru ϕ'_i luând în considerare toate combinaţiile de valori pe care le pot lua variabilele. Fiecare linie a tabelei de adevăr constă dintr-o combinaţie posibilă de valori şi valoarea expresiei pentru aceste valori. Folosind intrările tabelei de adevăr pentru care valoarea expresiei este 0, construim o formulă în ***formă normal-disjunctivă*** (sau ***FND***) – un SAU de ŞI-uri – formulă care este echivalentă cu $\neg\phi'_i$. Convertim apoi această formulă într-o formulă FNC ϕ''_i folosind regulile lui DeMorgan (5.2) pentru a complementa fiecare literal şi pentru a schimba ŞI-urile în SAU-uri şi SAU-urile în ŞI-uri.

În exemplul nostru convertim propoziţia $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ în FNC după cum urmează. Tabela de adevăr pentru ϕ'_1 este dată în figura 36.10. Formula FND echivalentă cu $\neg\phi'_1$ este

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2).$$

Aplicând legile lui DeMorgan, obţinem formula FNC

$$\phi''_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2),$$

care este echivalentă cu propoziţia iniţială ϕ'_1.

Fiecare propoziţie ϕ'_i a formulei ϕ' poate fi convertită în acelaşi mod într-o formulă FNC ϕ''_i, deci ϕ' este echivalentă cu formula FNC ϕ'' care constă din conjuncţia propoziţiilor ϕ''_i. Mai mult, fiecare propoziţie din ϕ'' are cel mult trei literali.

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Figura 36.10 Tabela de adevăr pentru propoziţia $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

La al treilea şi ultimul pas al reducerii vom transforma formula pentru ca fiecare propoziţie să aibă *exact* trei literali. Formula finală 3-FNC ϕ''' este construită cu ajutorul propoziţiilor formulei FNC ϕ''. Sunt folosite două variabile auxiliare pe care le vom denumi p şi q. Pentru fiecare propoziţie C_i din ϕ'' vom introduce în ϕ''' următoarele propoziţii:

- Dacă C_i are exact trei literali atunci ea va fi introdusă neschimbată în ϕ'''.
- Dacă C_i are doi literali distincţi, adică $C_i = (l_1 \vee l_2)$ unde l_1 şi l_2 sunt literali, atunci vom include în ϕ''' conjuncţia de propoziţii $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$. S-au introdus literalii p şi $\neg p$ pentru a putea fi îndeplinită cerinţa ca fiecare propoziţie să conţină exact 3 literali: $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ este echivalent cu $(l_1 \vee l_2)$ indiferent de valoarea lui p.
- Dacă C_i are un singur literal l, atunci vom include în ϕ''' conjuncţia de propoziţii $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$. Se observă că valoarea acestei expresii va fi l, indiferent de valorile lui p şi q.

Vom observa că formula 3-FNC ϕ''' este satisfiabilă dacă, şi numai dacă, ϕ este satisfiabilă. Asemeni reducerii de la CIRCUIT-SAT la SAT, construirea formulei ϕ' din ϕ nu alterează proprietatea formulei de a este sau nu satisfiabilă. La al doilea pas, se construieşte o formulă FNC ϕ'' care este echivalentă, din punct de vedere algebric, cu ϕ'. La al treilea pas, se construieşte o formulă 3-FNC ϕ''' care este echivalentă cu ϕ'', deoarece orice atribuire de valori pentru p şi q duce la o formulă echivalentă, din punct de vedere algebric, cu ϕ''.

Trebuie să arătăm şi faptul că această reducere poate fi efectuată în timp polinomial. La construirea formulei ϕ' din ϕ se introduce, cel mult, o variabilă şi o propoziţie pentru fiecare legătură din ϕ. La construirea formulei ϕ'' din ϕ' se introduc, cel mult, 8 propoziţii în ϕ'' pentru fiecare propoziţie din ϕ', deoarece fiecare propoziţie din ϕ' are cel mult 3 variabile, deci tabela de adevăr va avea, cel mult, $2^3 = 8$ linii. La construirea formulei ϕ''' din ϕ'' se introduc, cel mult, 4 propoziţii în ϕ''' pentru fiecare propoziţie din ϕ''. În concluzie, mărimea formulei ϕ''' depinde, după o funcţie polinomială, de lungimea formulei originale, fiecare construcţie putând fi realizată foarte uşor în timp polinomial. ■

Exerciţii

36.4-1 Se consideră reducerea în timp suprapolinomial prezentată în demonstraţia teoremei 36.9. Descrieţi un circuit de mărime n care, atunci când este convertit la o formulă folosind această metodă, duce la o formulă a cărei mărime depinde după o funcţie exponenţială de n.

36.4-2 Găsiţi formula 3-FNC care rezultă din formula (36.3) atunci când folosim metodele arătate în demonstraţia teoremei 36.10.

36.4-3 Profesorul Jagger propune o demonstraţie a faptului că SAT $\leq_P$ 3-FNC-SAT folosind numai metoda tabelei de adevăr din demonstraţia teoremei 36.10, fără să mai fie nevoie de ceilalţi paşi. Profesorul propune să luăm în considerare o formulă logică ϕ, să construim tabela de adevăr pentru variabilele sale, să găsim cu ajutorul acestei tabele o formulă 3-FND care este echivalentă cu $\neg\phi$ şi apoi, aplicând regulile lui DeMorgan, să obţinem o formulă 3-FNC echivalentă cu ϕ. Arătaţi că această strategie nu implică o reducere în timp polinomial.

36.4-4 Arătaţi că problema determinării faptului că o formulă logică este sau nu o tautologie este completă pentru co-NP. (*Indicaţie:* Vezi exerciţiul 36.3-6.)

36.4-5 Arătaţi că problema satisfiabilităţii formulelor în formă normal-disjunctivă este rezolvabilă în timp polinomial.

36.4-6 Să presupunem că cineva găseşte un algoritm polinomial pentru problema satisfiabilităţii formulei. Arătaţi cum poate fi folosit acest algoritm pentru a găsi atribuiri satisfiabile în timp polinomial.

36.4-7 Fie 2-FNC-SAT mulţimea formulelor logice, în formă normal-conjuctivă, care sunt satisfiabile şi care conţin exact 2 literali în fiecare propoziţie. Arătaţi că 2-FNC-SAT $\in$ P. Construiţi un algoritm cât mai eficient posibil. (*Indicaţie:* Observaţi că $x \vee y$ este echivalent cu $\neg x \rightarrow y$. Reduceţi problema 2-FNC-SAT la o problemă de grafuri orientate care este rezolvabilă eficient.)

36.5. Probleme NP-complete

Problemele NP-complete apar în cele mai diverse domenii: logică, grafuri, aritmetică, proiectarea reţelelor, mulţimi şi partiţii, memorări şi căutări, planificări, programarea matematică, algebră şi teoria numerelor, jocuri, teoria limbajelor şi a automatelor, optimizarea programelor şi multe altele. În această parte vom folosi metoda reducerii pentru a demonstra NP-completitudinea pentru o varietate de probleme din domeniul teoriei grafurilor şi a partiţionării mulţimilor.

Figura 36.11 ilustrează structura demonstraţiilor de NP-completitudine din această secţiune şi din secţiunea 36.4. Fiecare limbaj din figură este demonstrat a fi NP-complet printr-o reducere de la limbajul care îi este părinte în arborele prezentat. Rădăcina este CIRCUIT-SAT a cărui NP-completitudine a fost dovedită în teorema 36.7.

36.5.1. Problema clicii

O ***clică***, într-un graf neorientat $G = (V, E)$, este o submulţime $V' \subseteq V$ cu proprietatea că între fiecare pereche de vârfuri din V' există o muchie din E. Cu alte cuvinte, o clică este un subgraf complet al lui G. ***Mărimea*** clicii este dată de numărul de vârfuri pe care îl conţine. ***Problema clicii*** este problema de optim care cere găsirea unei clici de dimensiune maximă în graf. Problema de decizie corespunzătoare va fi determinarea faptului dacă există o clică de

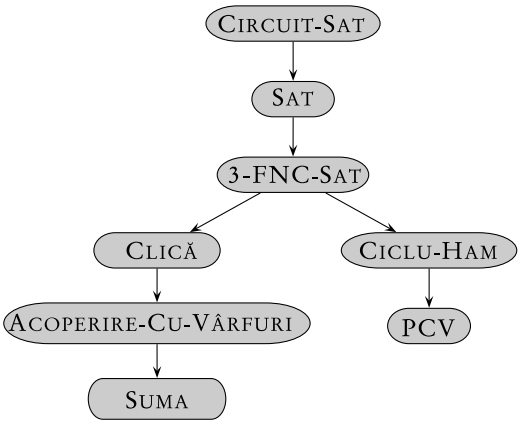


Figura 36.11 Structura demonstraţiilor de NP-completitudine din secţiunile 36.4 şi 36.5. Fiecare demonstraţie rezultă, în cele din urmă, din NP-completitudinea problemei satisfiabilităţii circuitului.

mărime k. Definiţia limbajului formal este: CLICĂ $= \{\langle G, k\rangle : G$ este un graf care conţine o clică de dimensiune $k\}$.

Un algoritm naiv pentru rezolvarea acestei probleme este de a determina toate submulţimile având k elemente ale lui V şi a verifica dacă formează o clică. Timpul de execuţie al acestui algoritm este $\Omega(k^2 \binom{|V|}{k})$, deci este un timp polinomial dacă k este o constantă. În general, k ar putea fi proporţional cu $|V|$, caz în care programul, care implementează algoritmul ar rula în timp suprapolinomial. După cum, probabil, presupuneţi este puţin probabil să existe un algoritm eficient pentru problema clicii.

Teorema 36.11 Problema clicii este NP-completă.

Demonstraţie. Pentru a arăta că CLICĂ $\in NP$ pentru un graf dat $G = (V, E)$, folosim mulţimea $V' \subseteq V$ a vârfurilor clicii ca probă pentru G. Verificarea faptului că V' este o clică poate fi efectuată în timp polinomial, verificând pentru fiecare pereche $u, v \in V'$ dacă muchia $(u, v) \in E$.

Arătăm, apoi, că problema clicii este NP-dificilă demonstrând că 3-FNC-SAT $\leq_P$ CLICĂ. Faptul că vom putea demonstra acest lucru este, într-un anume fel, surprinzător, deoarece formulele logice par a avea puţine în comun cu grafurile.

Algoritmul de reducere începe cu o instanţă a problemei 3-FNC-SAT. Fie $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ o formulă logică în 3-FNC cu k propoziţii. Pentru $r = 1, 2, \ldots, k$, fiecare propoziţie C_r are exact trei literali distincţi l_1^r, l_2^r şi l_3^r. Vom construi un graf G astfel încât ϕ să fie satisfiabilă dacă, şi numai dacă, G are o clică de mărime k.

Graful $G = (V, E)$ este construit după cum urmează. Pentru fiecare propoziţie $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ din ϕ introducem un triplet de vârfuri v_1^r, v_2^r şi v_3^r în V. Între două vârfuri v_i^r şi v_j^s va exista muchie dacă sunt îndeplinite simultan următoarele condiţii:

- v_i^r şi v_j^s fac parte din triplete diferite, adică $r \neq s$;
- literalii corespunzători acestor vârfuri sunt ***consistenţi***, adică l_i^r nu este negaţia lui l_j^s.

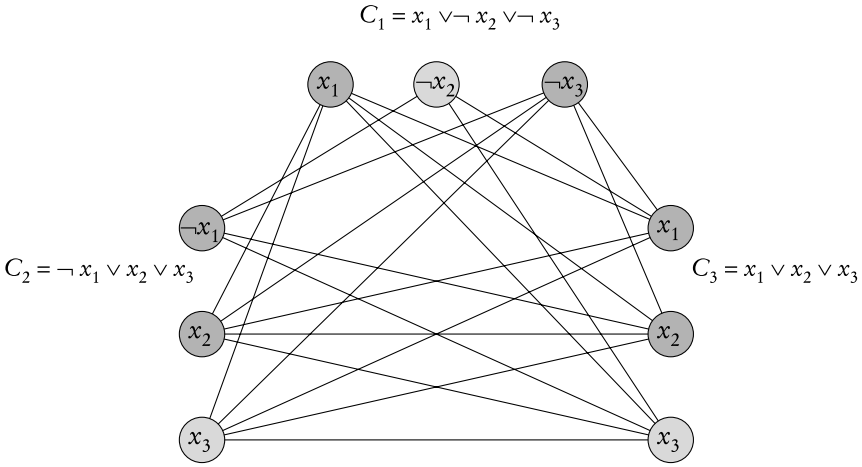


Figura 36.12 Graful G obţinut din formula 3-FNC $\phi = C_1 \wedge C_2 \wedge C_3$ unde $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3), C_2 = (\neg x_1 \vee x_2 \vee x_3)$ şi $C_3 = (x_1 \vee x_2 \vee x_3)$ pentru reducerea de la 3-FNC-SAT la CLICĂ. O atribuire satisfiabilă pentru formulă este $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. Această atribuire satisface C_1 prin $\neg x_2$, C_2 şi C_3 prin x_3. Vârfurile clicii corespunzătoare sunt cele deschise la culoare.

Acest graf poate fi foarte uşor determinat, cu ajutorul lui ϕ, în timp polinomial. În figura 36.12 este prezentat graful corespunzător expresiei:

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3).$$

Trebuie să arătăm că această transformare de la ϕ la G este, într-adevăr, o reducere. Să presupunem, mai întâi, că ϕ are o atribuire satisfiabilă. Atunci, fiecare propoziţie C_r conţine cel puţin un literal l_i^r căruia i-a fost atribuită valoarea 1 şi fiecare astfel de literal corespunde unui vârf v_i^r. Alegând un astfel de literal pentru fiecare propoziţie, obţinem o mulţime de k vârfuri. Afirmăm că V' este o clică. Pentru fiecare două vârfuri v_i^r şi $v_j^s \in V'$, cu $r \neq s$, literalilor corespunzători le-a fost atribuită valoarea 1, deci aceşti literali nu pot fi unul complementul celuilalt. Deci, datorită modului în care a fost construit graful G, muchia (v_i^r, v_j^s) aparţine lui E.

Invers, să presupunem că G are o clică V' de dimensiune k. Nici o muchie din G nu leagă vârfuri din acelaşi triplet, deci V' conţine exact un vârf pentru fiecare triplet. Putem atribui valoarea 1 pentru fiecare literal l_i^r, pentru care $v_i^r \in V'$, fără a mai fi necesară verificarea cazului în care a fost atribuită valoarea 1, atât unui literal cât şi complementului său, deoarece G nu conţine muchii între vârfurile corespunzătoare unor literali inconsistenţi. Astfel, fiecare propoziţie va fi satisfiabilă, deci ϕ va fi şi ea satisfiabilă. (Fiecare variabilă care nu corespunde vreunui vârf al grafului poate primi o valoare arbitrară.) ■

În exemplul din figura 36.12 o atribuire satisfiabilă pentru ϕ este $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$, iar clica de dimensiune $k = 3$ corespunzătoare constă din vârfurile care corespund lui $\neg x_2$ din prima propoziţie, x_3 din a doua şi tot x_3 din a treia.

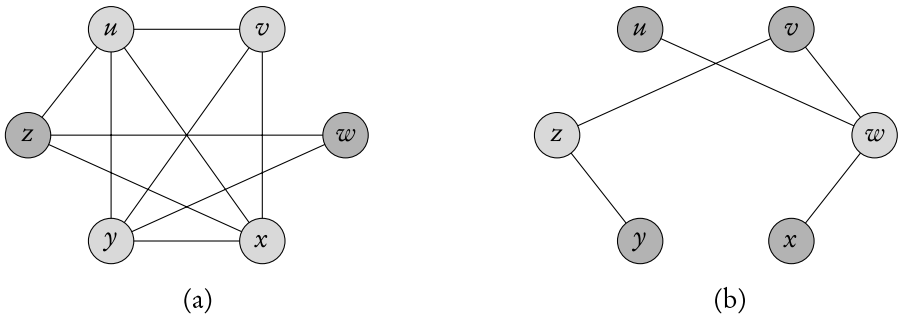


Figura 36.13 Reducere de la CLICĂ la ACOPERIRE-CU-VÂRFURI. **(a)** Un graf neorientat $G = (V, E)$ cu clica $V' = \{u, v, x, y\}$ **(b)** Graful $\overline{G}$ obţinut în urma algoritmului de reducere care are mulţimea stabilă $V - V' = \{w, z\}$.

36.5.2. Problema acoperirii cu vârfuri

O ***acoperire cu vârfuri*** a unui graf neorientat $G = (V, E)$ este o submulţime $V' \subseteq V$ astfel încât, dacă $(u, v) \in E$, atunci cel puţin unul din vârfurile u şi v face parte din V'. Aceasta înseamnă că fiecare vârf îşi "acoperă" muchiile incidente şi că o acoperire cu vârfuri pentru G este o mulţime de vârfuri care acoperă toate muchiile din E. ***Dimensiunea*** unei acoperiri cu vârfuri este dată de numărul vârfurilor pe care le conţine. De exemplu, graful din figura 36.13(b) are o acoperire cu vârfuri $\{w, z\}$ de dimensiune 2.

Problema acoperirii cu vârfuri cere determinarea unei acoperiri cu vârfuri de dimensiune minimă într-un graf dat. Reformulând această problemă de optim ca o problemă de decizie, vom cere verificarea faptului că un graf are o acoperire cu vârfuri de dimensiune dată k. Limbajul formal corespunzător poate fi definit astfel:

ACOPERIRE-CU-VÂRFURI $= \{\langle G, k\rangle : \text{graful } G \text{ are o acoperire cu vârfuri de dimensiune } k\}$. Următoarea teoremă arată că această problemă este NP-completă.

Teorema 36.12 Problema acoperirii cu vârfuri este NP-completă.

Demonstraţie. Arătăm, mai întâi, că ACOPERIRE-CU-VÂRFURI $\in$ NP. Să presupunem că avem un graf $G = (V, E)$ şi un întreg k. Proba pe care o alegem este chiar acoperirea cu vârfuri $V' \subseteq V$. Algoritmul de verificare afirmă că $|V'| = k$ şi apoi verifică, pentru fiecare muchie $(u, v) \in E$, dacă $u \in V'$ sau $v \in V'$. Această verificare poate fi efectuată în mod direct, în timp polinomial.

Demonstrăm că problema acoperirii cu vârfuri este NP-dificilă arătând că CLICĂ $\leq_P$ ACOPERIRE-CU-VÂRFURI. Reducerea este bazată pe noţiunea de "complement" al unui graf. Dându-se un graf neorientat $G = (V, E)$, definim ***complementul*** lui G ca fiind $\overline{G} = (V, \overline{E})$ unde $\overline{E} = \{(u, v) : (u, v) \notin E\}$. Cu alte cuvinte $\overline{G}$ este graful care conţine exact acele muchii care nu sunt conţinute în G. În figura 36.13 se arată un graf şi complementul său şi ilustrează reducerea de la CLICĂ la ACOPERIRE-CU-VÂRFURI.

Algoritmul de reducere are ca intrare o instanţă $\langle G, k\rangle$ a problemei clicii. Calculează complementul $\overline{G}$, fapt care poate fi realizat foarte uşor în timp polinomial. Ieşirea algoritmului de reducere este instanţa $\langle \overline{G}, |V| - k\rangle$ a problemei acoperirii cu vârfuri. Pentru a completa demonstraţia, arătăm că această transformare este, într-adevăr, o reducere: graful G are o clică de dimensiune k dacă, şi numai dacă, graful $\overline{G}$ are o acoperire cu vârfuri de dimensiune $|V| - k$.

Să presupunem că G are o clică $V' \subseteq V$ cu $|V'| = k$. Susţinem că $V - V'$ este o acoperire cu vârfuri în $\overline{G}$. Fie (u, v) o muchie în $\overline{E}$. Atunci $(u, v) \notin E$, ceea ce implică faptul că cel puţin unul din vârfurile u şi v nu aparţine lui V' deoarece fiecare pereche de vârfuri din V' este legată printr-o muchie din E. Acest lucru este echivalent cu faptul că cel puţin unul din vârfurile u şi v este în $V - V'$, ceea ce înseamnă că muchia (u, v) este acoperită de $V - V'$. Deoarece (u, v) a fost aleasă în mod arbitrar din $\overline{E}$, fiecare muchie din $\overline{E}$ este acoperită de un vârf din $V - V'$. În concluzie, mulţimea $V - V'$, care are dimensiunea $|V| - k$, este o acoperire cu vârfuri pentru $\overline{G}$.

Invers, să presupunem că $\overline{G}$ are o acoperire cu vârfuri $V' \subseteq V$, unde $|V'| = |V| - k$. Pentru orice $u, v \in V$, dacă $(u, v) \in \overline{E}$, atunci cel puţin unul din vârfurile u şi v aparţine lui V'. O implicaţie echivalentă este că pentru orice $u, v \in V$ dacă, $u \notin V'$ şi $v \notin V'$, atunci, $(u, v) \in E$. Cu alte cuvinte $V - V'$ este o clică de dimensiune $|V| - |V'| = k$. ∎

Deoarece ACOPERIRE-CU-VÂRFURI este NP-completă, nu ne aşteptăm să găsim un algoritm polinomial pentru determinarea unei acoperiri cu vârfuri minimale. În secţiunea 37.1 este prezentat un "algoritm de aproximare" polinomial care dă o soluţie "aproximativă" pentru problema acoperirii cu vârfuri. Dimensiunea acoperirii cu vârfuri, furnizată de acest algoritm, este cel mult de două ori mai mare decât dimensiunea minimă a acoperirii cu vârfuri.

În concluzie, nu ar trebui să încetăm să fim optimişti doar din cauză că o problemă este NP-completă. Poate exista un algoritm de aproximare polinomial care obţine o soluţie aproape optimă, chiar dacă găsirea unei soluţii optime este o problemă NP-completă. În capitolul 37 sunt prezentaţi mai mulţi algoritmi de aproximare pentru diferite probleme NP-complete.

36.5.3. Problema sumei submulţimii

Următoarea problemă NP-completă pe care o vom lua în considerare este o problemă de aritmetică. În ***problema sumei submulţimii*** se dă o mulţime finită $S \subset \mathbb{N}$ şi o ţintă $t \in \mathbb{N}$ şi se cere verificarea faptului că există o submulţime S' a lui S, pentru care suma elementelor componente este t. De exemplu, dacă $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ şi $t = 3754$ atunci submulţimea $S' = \{1, 16, 64, 256, 1040, 1093, 1284\}$ este o soluţie.

Ca de obicei vom defini problema ca un limbaj formal:

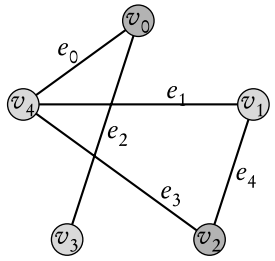
$$\text{SUMA} = \{\langle S, t\rangle : \text{există o submulţime } S' \subseteq S \text{ astfel încât } t = \sum_{s \in S'} s\}.$$

Ca pentru orice problemă de aritmetică, este important să amintim că o codificare standard presupune că numerele întregi date la intrare sunt reprezentaţi în baza 2. Ţinând cont de această presupunere putem arăta că este puţin probabil ca problema sumei submulţimii să aibă un algoritm rapid.

Teorema 36.13 Problema sumei submulţimii este NP-completă.

Demonstraţie. Pentru a arăta că SUMA $\in NP$, pentru fiecare instanţă $\langle S, t\rangle$ a acestei probleme, luăm ca probă submulţimea S'. Algoritmul care verifică dacă $t = \sum_{s \in S'} s$ este unul polinomial.

Arătăm, acum, că ACOPERIRE-CU-VÂRFURI $\leq_P$ SUMA. Dându-se o instanţă $\langle G, k\rangle$ a problemei acoperirii cu vârfuri, algoritmul de reducere va construi o instanţă $\langle S, t\rangle$ a problemei sumei submulţimii astfel încât G va avea o acoperire cu vârfuri de dimensiune k dacă şi numai, dacă, există o submulţime a lui S pentru care suma elementelor este exact t.



(a)

	e_4	e_3	e_2	e_1	e_0
v_0	0	0	1	0	1
v_1	1	0	0	1	0
v_2	1	1	0	0	0
v_3	0	0	1	0	0
v_4	0	1	0	1	1

(b)

		baza 4 modificată						zecimal
		e_4	e_3	e_2	e_1	e_0		
x_0	= 1	0	0	1	0	1	=	1041
x_1	= 1	1	0	0	1	0	=	1284
x_2	= 1	1	1	0	0	0	=	1344
x_3	= 1	0	0	1	0	0	=	1040
x_4	= 1	0	1	0	1	1	=	1093
y_0	= 0	0	0	0	0	1	=	1
y_1	= 0	0	0	0	1	0	=	4
y_2	= 0	0	0	1	0	0	=	16
y_3	= 0	0	1	0	0	0	=	64
y_4	= 0	1	0	0	0	0	=	256
t	= 3	2	2	2	2	2	=	3754

(c)

Figura 36.14 Reducerea problemei acoperirii cu vârfuri la problema sumei submulţimii. **(a)** Un graf neorientat G. O acoperire cu vârfuri $\{v_1, v_3, v_4\}$ de dimensiune 3, mulţime formată din vârfurile haşurate deschis. **(b)** Matricea de incidenţă corespunzătoare. Liniile haşurate deschis corespund acoperirii cu vârfuri de la (a). Fiecare muchie e_j are un 1 în cel puţin o linie haşurată deschis. **(c)** Instanţa corespunzătoare pentru problema sumei submulţimii. Porţiunea din chenar este matricea de incidenţă. Aici acoperirea cu vârfuri $\{v_1, v_3, v_4\}$ de dimensiune $k = 3$ corespunde submulţimii haşurate deschis $\{1, 16, 64, 256, 1040, 1093, 1284\}$ pentru care suma elementelor este 3754.

Elementul de bază al reducerii este o reprezentare a lui G cu ajutorul unei matrice de incidenţă. Fie $G = (V, E)$ un graf neorientat, $V = \{v_0, v_1, \ldots, v_{|V|-1}\}$ şi $E = \{e_0, e_1, \ldots, e_{|E|-1}\}$. ***Matricea de incidenţă*** a lui G este o matrice $B = (b_{ij})$, de dimensiune $|V| \times |E|$, astfel încât

$$b_{ij} = \begin{cases} 1 & \text{dacă muchia } e_j \text{ este incidentă vârfului } v_i, \\ 0 & \text{în caz contrar.} \end{cases}$$

De exemplu, figura 36.14(b) arată matricea de incidenţă pentru graful neorientat din figura 36.14(a). Matricea de incidenţă conţine muchiile cu indicele mai mic la dreapta şi nu la stânga, aşa cum este reprezentarea convenţională. S-a făcut această alegere pentru a simplifica formulele pentru numerele din S.

Dându-se un graf G şi un întreg k, algoritmul de reducere determină o mulţime S de numere şi un întreg t. Pentru a înţelege cum funcţionează acest algoritm, reprezentăm numerele în "baza 4 modificată". Cele $|E|$ cifre de ordin inferior vor fi cifre ale bazei 4, iar cele de ordin superior pot lua valori până la k. Mulţimea de numere este construită în aşa fel, încât nici un transport nu se poate propaga de la cifrele de ordin inferior la cele de ordin superior.

Mulţimea S constă din două tipuri de numere corespunzând vârfurilor, respectiv muchiilor. Pentru fiecare vârf $v_i \in V$, creăm un întreg pozitiv x_i a cărui reprezentare în baza 4 modificată constă dintr-un 1 urmat de $|E|$ cifre. Cifrele corespund celei de-a v_i-a linie a matricei de incidenţă $B = (b_{ij})$ a lui G, după cum se vede în figura 36.14(c). Putem spune că pentru $i = 0, 1, \ldots, |V|-1$

avem

$$x_i = 4^{|E|} + \sum_{j=0}^{|E|-1} b_{ij} 4^j.$$

Pentru fiecare muchie $e_j \in E$, creăm un întreg pozitiv y_j care reprezintă o linie a matricei "identitate" de incidenţă. (Matricea identitate de incidenţă este o matrice de dimensiuni $|E| \times |E|$ cu elemente egale cu 1 numai pe poziţiile de pe digonală. Putem spune că pentru $j = 0, 1, \ldots, |E| - 1$

$$y_j = 4^j.$$

Prima cifră a sumei ţintă t este k şi celelalte $|E|$ cifre de ordin inferior sunt 2-uri. Putem spune că

$$t = k4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^j.$$

Toate aceste numere au mărime polinomială când sunt reprezentate în baza 2. Reducerea poate fi efectuată în timp polinomial, manipulând biţii matricei de incidenţă.

Trebuie să arătăm că graful G are o acoperire cu vârfuri de dimensiune k dacă, şi numai dacă, există o submulţime $S' \subseteq S$ pentru care suma elementelor este t. Să presupunem, mai întâi, că G are o acoperire cu vârfuri $V' \subseteq V$ de dimensiune k. Fie $V' = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ şi $S' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \cup \{y_j : e_j \text{ este incidentă exact unui vârf din } V'\}$.

Pentru a arăta că $\sum_{s \in S'} s = t$, să observăm că, însumând cei k de 1 cu care începe fiecare $x_{i_m} \in S'$, obţinem cifra k ca primă cifră pentru reprezentarea în baza 4 modificată a lui t. Pentru a obţine cifrele de ordin inferior ale lui t (toate sunt 2), să considerăm, pe rând, poziţiile cifrelor, fiecare corespunzând unei muchii e_j. Deoarece V' este o acoperire cu vârfuri, e_j este incidentă cel puţin unui vârf din V'. Deci, pentru fiecare muchie e_j, există cel puţin un $x_{i_m} \in S'$ cu 1 pe poziţia j. Dacă e_j este incidentă la două vârfuri din V', atunci ambele vârfuri contribuie cu un 1 în poziţia j. A j-a cifră a lui y_j nu contribuie cu nimic deoarece e_j este incidentă la două vârfuri, ceea ce implică $y_j \notin S'$. Deci, în acest caz, suma pentru S' va duce la apariţia cifrei 2 pe poziţia j a lui t. În celălalt caz – când e_j este incidentă exact unui vârf din V' – avem $y_j \in S'$, iar vârful incident şi y_j vor contribui ambele cu câte un 1 pe poziţia j a lui t, deci va apărea tot un 2. Deci S' este o soluţie pentru instanţa S a problemei sumei submulţimii.

Să presupunem, acum, că există o submulţime $S' \subseteq S$ pentru care suma elementelor este t. Fie $S' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\} \cup \{y_{j_1}, y_{j_2}, \ldots, y_{j_p}\}$. Afirmăm că $m = k$ şi că $V' = \{v_{i_1}, v_{i_2}, \ldots, v_{i_m}\}$ este o acoperire cu vârfuri pentru G. Pentru a demonstra acest lucru, începem prin a observa că, pentru fiecare muchie $e_j \in E$, S conţine trei întregi care au cifra 1 pe poziţia e_j: câte unul pentru fiecare dintre vârfurile incidente muchiei e_j şi unul pentru y_j. Deoarece lucrăm cu reprezentarea în baza 4 modificată, nu vor fi transferuri de la poziţia e_j la poziţia e_{j+1}. Deci, pentru fiecare din cele $|E|$ poziţii de ordin inferior ale lui t, cel puţin unul şi cel mult doi x_i trebuie să contribuie la sumă. Deoarece ultimul x_i contribuie la suma corespunzătoare fiecărei muchii, observăm că V' este o acoperire cu vârfuri. Pentru a demonstra că $m = k$ şi, deci, că V' este o acoperire cu vârfuri de dimensiune k, observăm că singura modalitate de a obţine k pe prima poziţie a ţintei t este a include x_i de k ori în sumă. ∎

În figura 36.14, acoperirea cu vârfuri $V' = \{v_1, v_3, v_4\}$ corespunde submulţimii $S' = \{x_1, x_3, x_4, y_0, y_2, y_3, y_4\}$. Fiecare y_j este inclus în S' cu excepţia lui y_1 care este incident la două vârfuri din V'.

36.5.4. Problema ciclului hamiltonian

Ne întoarcem la problema ciclului hamiltonian definită în secţiunea 36.2.

Teorema 36.14 Problema ciclului hamiltonian este NP-completă.

Demonstraţie. Arătăm, mai întâi, că CICLU-HAM $\in$ NP. Dându-se un graf $G = (V, E)$, proba este o secvenţă de $|V|$ noduri care formează ciclul hamiltonian. Verificăm, cu ajutorul algoritmului de verificare, dacă secvenţa conţine fiecare vârf din V exact o dată şi că primul vârf se repetă la sfârşit. Această verificare poate fi făcută în timp polinomial.

Demonstrăm că CICLU-HAM este NP-completă arătând că 3-FNC-SAT $\leq_P$ CICLU-HAM. Dându-se o formulă logică ϕ în formă normală conjunctivă care conţine variabilele $x_1, x_2, \ldots, x_n$ cu propoziţiile $C_1, C_2, \ldots, C_k$, fiecare conţinând exact 3 literali distincţi, construim graful $G = (V, E)$ în timp polinomial astfel încât G are un ciclu hamiltonian dacă, şi numai dacă, ϕ este satisfiabilă. Construcţia noastră se bazează pe ***grafuri ajutătoare*** care sunt componente ale grafurilor care au câteva proprietăţi speciale.

Primul graf ajutător este subgraful A prezentat în figura 36.15(a). Să presupunem că A este un subgraf al unui graf G şi singurele legături dintre A şi restul grafului G sunt realizate prin nodurile a, a', b şi b'. Să presupunem că G are un ciclu hamiltonian. Deoarece orice ciclu hamiltonian al lui G trebuie să treacă prin vârfurile z_1, z_2, z_3 şi z_4 într-unul din modurile arătate în figurile 36.15(b) şi (c), putem trata subgraful A ca şi cum ar fi o simplă pereche de muchii (a, a') şi (b, b') cu restricţia că orice ciclu hamiltonian din G trebuie să includă exact una dintre aceste muchii. Vom reprezenta graful ajutător A aşa cum este arătat în figura 36.15(d).

Subgraful B din figura 36.16 este al doilea graf ajutător pe care îl luăm în considerare. Să presupunem că B este un subgraf al unui graf G şi că singurele legături dintre B şi restul grafului G sunt prin nodurile b_1, b_2, b_3 şi b_4. Un ciclu hamiltonian al grafului G nu poate conţine, concomitent, muchiile (b_1, b_2), (b_2, b_3) şi (b_3, b_4) deoarece, în acest caz, toate celelalte vârfuri ale grafului ajutător, în afară de b_1, b_2, b_3 şi b_4, nu vor putea apărea în ciclu. Un ciclu hamiltonian al lui G poate traversa orice submulţime a acestor muchii. Figurile 36.16 (a)–(e) arată cinci astfel de submulţimi; cele două submulţimi rămase pot fi obţinute prin oglindirea faţă de orizontală a părţilor (b) şi (e). Reprezentăm acest graf ajutător ca în figura 36.18(f), ideea fiind că cel puţin unul din drumurile indicate de săgeţi trebuie să apară şi în ciclul hamiltonian.

Graful G pe care îl vom construi constă, în cea mai mare măsură, din aceste două grafuri ajutătoare. Construcţia este ilustrată în figura 36.17. Pentru fiecare din cele k propoziţii din ϕ, includem o copie a grafului ajutător B şi unim aceste grafuri ajutătoare într-o serie după cum urmează. Dacă b_{ij} este copia vârfului b_j în a i-a componentă a lui B, conectăm $b_{i,4}$ cu $b_{i+1,1}$ pentru $i = 1, 2, \ldots, k-1$.

Apoi, pentru fiecare variabilă x_m din ϕ, includem două vârfuri x'_m şi x''_m. Conectăm aceste două vârfuri prin două copii ale muchiei (x'_m, x''_m) pe care le notăm prin e_m şi $\overline{e}_m$ pentru a le distinge. Ideea este că, dacă ciclul hamiltonian trece prin e_m, aceasta corespunde atribuirii valorii 1 variabilei x_m. Dacă ciclul hamiltonian trece prin muchia $\overline{e}_m$, valoarea atribuită variabilei x_m este 0. Fiecare pereche de muchii formează un ciclu având două muchii. Conectăm aceste mici

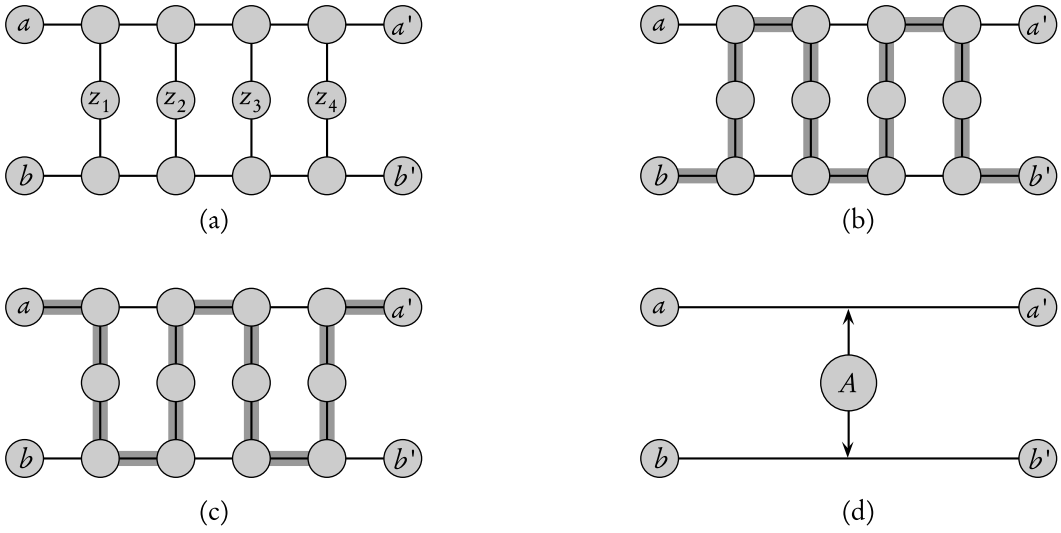


Figura 36.15 **(a)** Graful ajutător A folosit în reducerea de la 3-FNC-SAT la CICLU-HAM. **(b)-(c)** Dacă A este un subgraf al unui graf G care conţine un ciclu hamiltonian şi singurele legături ale lui A cu restul grafului G se fac prin vârfurile a, a', b şi b', atunci, muchiile haşurate reprezintă singurele drumuri posibile în care ciclul hamiltonian poate traversa muchiile subgrafului A. **(d)** O reprezentare compactă a grafului ajutător A.

cicluri într-o serie adăugând muchiile (x'_m, x''_{m+1}) pentru $m = 1, 2, \ldots, n-1$. Legăm partea dreaptă a grafului cu partea stângă cu ajutorul a două muchii $(b_{1,1}, x'_1)$ şi $(b_{k,4}, x''_n)$, care reprezintă muchiile cele mai de sus şi cele mai de jos din figura 36.17.

Încă nu am terminat construcţia grafului G deoarece trebuie să realizăm legătura dintre variabile şi propoziţii. Dacă cel de-al j-lea literal al propoziţiei C_i este x_m, atunci folosim un graf ajutător A pentru a uni muchia $(b_{i,j}, b_{i,j+1})$ cu muchia e_m. Dacă cel de-al j-lea literal al propoziţiei C_i este $\neg x_m$, atunci folosim un graf ajutător A pentru a uni muchia $(b_{i,j}, b_{i,j+1})$ cu muchia $\overline{e}_m$. În figura 36.17, de exemplu, deoarece propoziţia C_2 este $(x_1 \vee \neg x_2 \vee x_3)$, plasăm trei grafuri ajutătoare A după cum urmează:

- între $(b_{2,1}, b_{2,2})$ şi e_1,
- între $(b_{2,2}, b_{2,3})$ şi $\overline{e}_2$,
- între $(b_{2,3}, b_{2,4})$ şi e_3.

Se observă că, pentru conectarea a două muchii printr-un graf ajutător A, trebuie să înlocuim fiecare muchie cu cele cinci muchii din partea de sus sau de jos a figurii 36.15(a) şi, bineînţeles, să adăugăm legăturile care trec prin cele z vârfuri. Un literal dat l_m poate apărea în mai multe propoziţii ($\neg x_3$ din figura 36.17, de exemplu), deci o muchie e_m sau $\overline{e}_m$ poate fi influenţată de mai multe grafuri ajutătoare A (muchia $\overline{e}_3$ de exemplu). În acest caz, conectăm grafurile ajutătoare A în serie, aşa cum se arată în figura 36.18, înlocuind muchiile e_m sau $\overline{e}_m$ printr-o serie de muchii.

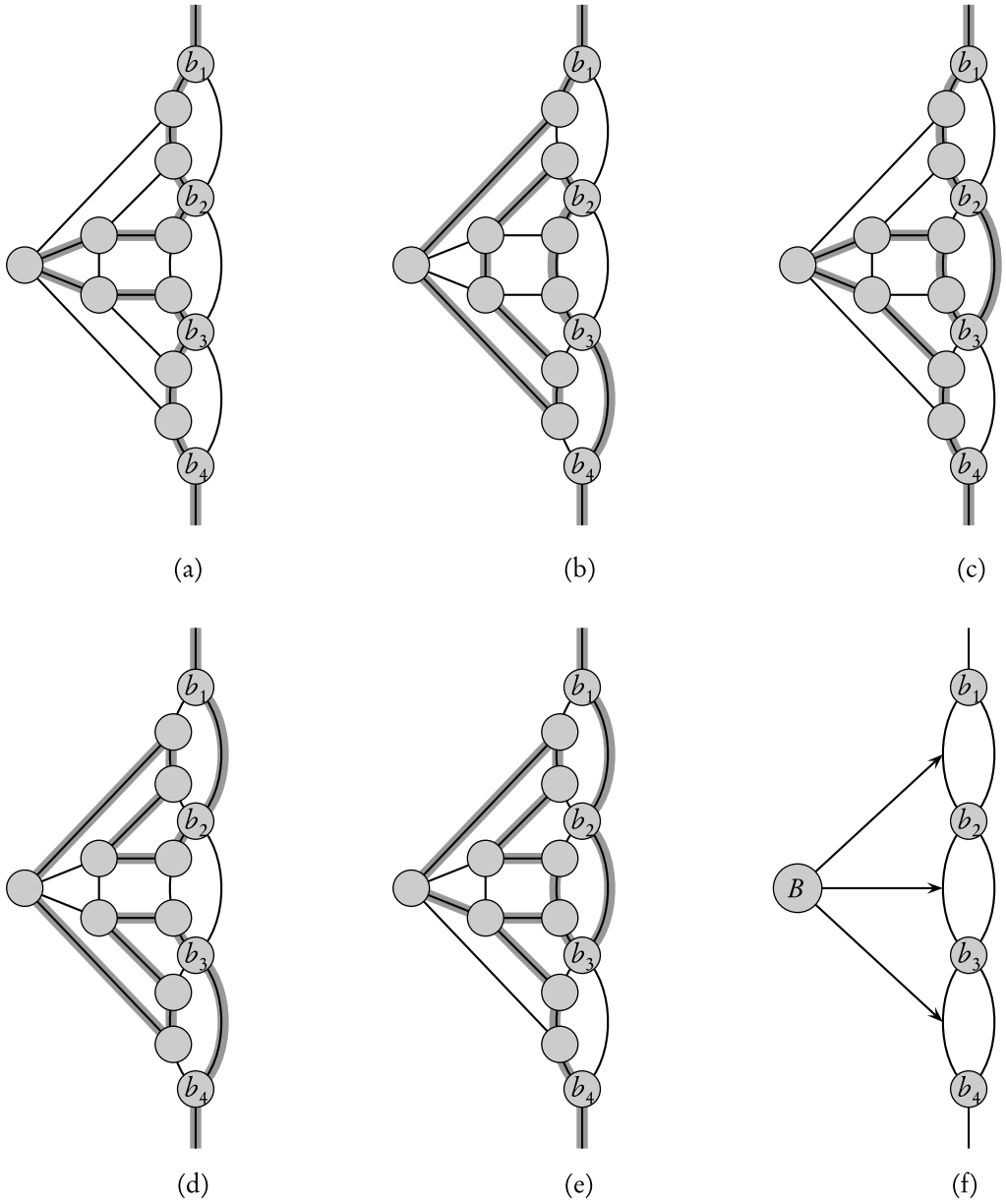


Figura 36.16 Graful ajutător B, folosit în reducerea de la 3-FNC-SAT la CICLU-HAM. Nici un drum de la vârful b_1 la vârful b_4, care conţine toate vârfurile din graful ajutător, nu poate folosi concomitent muchiile (b_1, b_2), (b_2, b_3) şi (b_3, b_4). Orice submulţime a acestei muchii poate fi folosită. **(a)-(e)** Cinci astfel de submulţimi. **(f)** O reprezentare a acestui graf ajutător în care cel puţin unul din drumurile indicate de săgeţi trebuie să apară în ciclul hamiltonian.

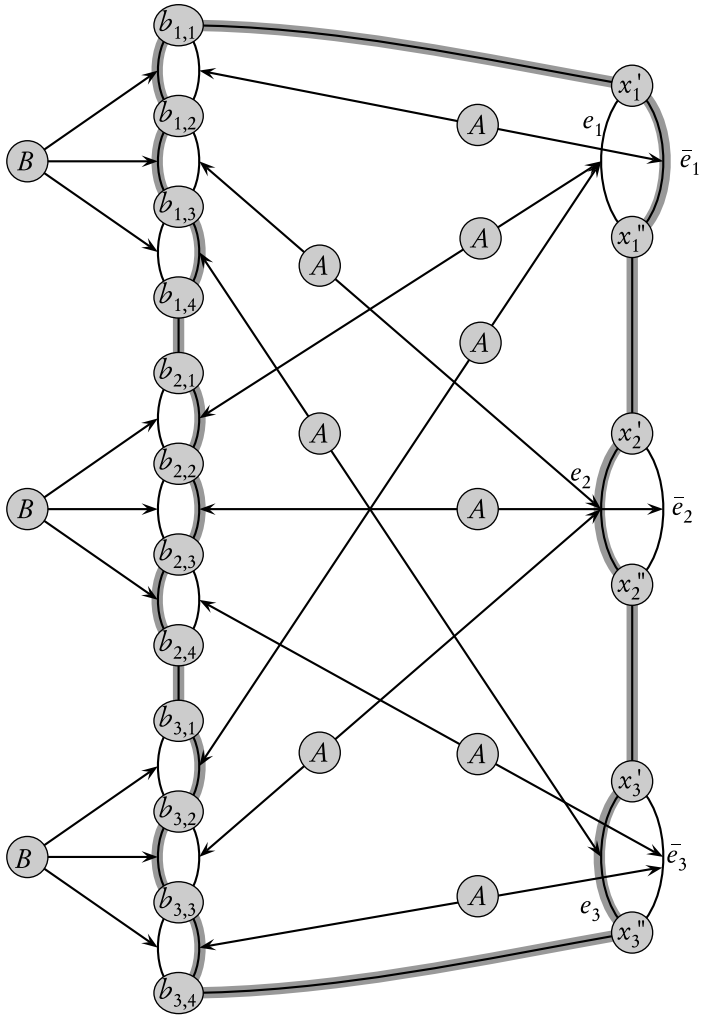


Figura 36.17 Graful G construit din formula $\phi = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. O atribuire satisfiabilă s pentru variabilele din ϕ este $s(x_1) = 0$, $s(x_2) = 1$ şi $s(x_3) = 1$, ceea ce corespunde ciclului hamiltonian prezentat. Observaţi că dacă $s(x_m) = 1$ atunci muchia e_m apare în ciclul hamiltonian şi dacă $s(x_m) = 0$ atunci în ciclul hamiltonian apare muchia $\overline{e}_m$.

Susţinem că formula ϕ este satisfiabilă dacă, şi numai dacă, graful G conţine un ciclu hamiltonian. Presupunem, mai întâi, că G are un ciclu hamiltonian h şi arătăm că ϕ este satisfiabilă. Ciclul h trebuie să aibă o formă particulară:

- Mai întâi traversează muchia $(b_{1,1}, x'_1)$ pentru a ajunge din stânga-sus în dreapta-sus.
- Urmează apoi toate vârfurile x'_m şi x''_m de sus în jos, alegând fie muchia e_m, fie muchia $\overline{e}_m$ dar nu ambele.

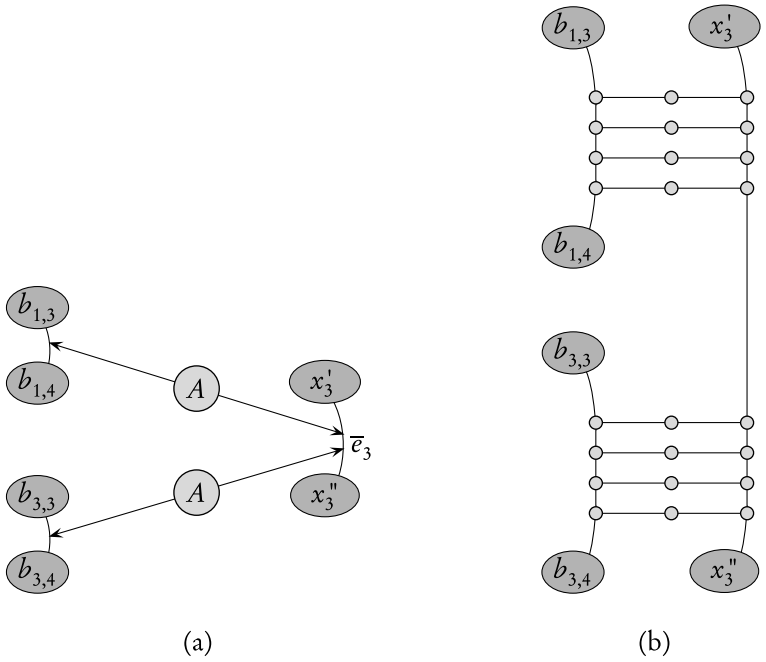


Figura 36.18 Construcţia folosită când o muchie e_m sau $\overline{e}_m$ este influenţată de mai multe grafuri ajutătoare A. **(a)** O porţiune a figurii 36.17. **(b)** Subgraful construit.

- Traversează muchia $(b_{k,4}, x_n'')$ pentru a ajunge în partea stângă.
- În final, traversează grafurile ajutătoare B de jos în sus.

(Traversează, de fapt, şi muchiile din grafurile ajutătoare A, dar folosim aceste subgrafuri pentru a scoate în evidenţă natura sau/sau a muchiilor pe care le leagă.)

Dându-se un ciclu hamiltonian h, definim o atribuire de adevăr pentru ϕ după cum urmează. Dacă $e_m \in h$ atunci atribuim valoarea 1 variabilei x_m. În caz contrar, muchia $\overline{e}_m \in h$ şi valoarea variabilei x_m va fi 0.

Susţinem că această atribuire satisface ϕ. Să considerăm propoziţia C_i şi graful ajutător B corespunzător din G. Fiecare muchie $(b_{i,j}, b_{i,j+1})$ este conectată printr-un graf ajutător A, fie la muchia $\overline{e}_m$, fie la muchia e_m, depinzând de faptul dacă al j-lea literal este x_m sau $\neg x_m$. Muchia $(b_{i,j}, b_{i,j+1})$ este traversată de h dacă, şi numai dacă, literalul corespunzător este 0. Deoarece fiecare din cele 3 muchii $(b_{i,1}, b_{i,2})$, $(b_{i,2}, b_{i,3})$ şi $(b_{i,3}, b_{4i,})$ ale propoziţiei C_i apare, de asemenea, într-un graf ajutător B, nu pot fi traversate toate trei de ciclul hamiltonian h. Una dintre cele trei muchii trebuie să aibă literalul corespunzător cu valoarea 1, deci propoziţia C_i este satisfiabilă. Această proprietate este respectată pentru fiecare propoziţie $C_i, i = 1, 2, \ldots, k$, deci formula ϕ este satisfiabilă.

Invers, să presupunem că formula ϕ este satisfiabilă folosind o anumită atribuire de adevăr. Urmărind regulile de deasupra putem construi un ciclu hamiltonian pentru graful G: traversăm muchia e_m dacă $x_m = 1$, muchia $\overline{e}_m$ dacă $x_m = 0$ şi muchia $(b_{i,j}, b_{i,j+1})$ dacă, şi numai dacă, valoarea atribuită celui de-al j-lea literal al propoziţiei C_i este 0. Aceste reguli pot fi într-adevăr

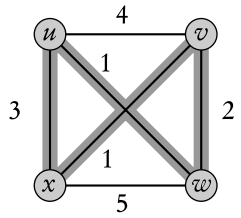


Figura 36.19 O instanţă a problemei comis-voiajorului. Muchiile haşurate reprezintă un ciclu de cost minim, costul fiind 7.

urmărite, deoarece presupunem că s este o atribuire satisfiabilă pentru formula ϕ.

În final, să observăm că graful G poate fi construit în timp polinomial. El conţine un graf ajutător B pentru fiecare dintre cele k propoziţii ale lui ϕ. Există un graf ajutător A pentru fiecare instanţă a fiecărui literal din ϕ, deci există $3k$ grafuri ajutătoare A. Deoarece grafurile ajutătoare A şi B au dimensiune fixă, graful G are $O(k)$ vârfuri şi muchii şi poate fi foarte uşor construit în timp polinomial. Putem spune, acum, că am furnizat o reducere în timp polinomial de la 3-FNC-SAT la CICLU-HAM. ■

36.5.5. Problema comis-voiajorului

În ***problema comis-voiajorului***, care este în strânsă legătură cu problema ciclului hamiltonian, un comis-voiajor trebuie să viziteze n oraşe. Modelând problema pe un graf cu n vârfuri, putem spune că comis-voiajorul doreşte să facă un ***tur***, sau un ciclu hamiltonian, vizitând fiecare oraş o singură dată şi terminând cu oraşul din care a pornit. Pentru călătoria între oraşele i şi j există un cost $c(i, j)$ reprezentat printr-un număr întreg. Comis-voiajorul doreşte să parcurgă un ciclu al cărui cost total să fie minim, unde costul total este suma costurilor individuale de pe muchiile care formează ciclul. De exemplu, în figura 36.19, un ciclu de cost minim este $\langle u, w, v, x, u\rangle$, acesta având costul 7. Limbajul formal pentru problema de decizie corespunzătoare este:

$$\text{PCV} = \{\langle G, c, k\rangle : \ G = (V, E) \text{ este un graf complet, } c \text{ este o funcţie de la } V \times V \to \mathbb{Z}, \\ k \in \mathbb{Z} \text{ şi } G \text{ are un ciclu pentru comisul voiajor, de cost cel mult } k\}.$$

Următoarea teoremă arată că este puţin probabil să existe un algoritm rapid pentru problema comis voiajorului.

Teorema 36.15 Problema comis-voiajorului este NP-completă.

Demonstraţie. Arătăm, mai întâi, că PCV $\in$ NP. Dându-se o instanţă a problemei, folosim ca probă şirul de n vârfuri care formează ciclul. Algoritmul de verificare testează dacă acest şir conţine fiecare nod exact o dată, adună costurile muchiilor şi verifică dacă suma este cel mult k. Acest lucru poate fi, cu siguranţă, realizat în timp polinomial.

Pentru a demonstra că PCV este NP-dificilă, arătăm că CICLU-HAM $\leq_{\text{P}}$ PCV. Fie $G = (V, E)$ o instanţă a problemei ciclului hamiltonian. Construim o instanţă a PCV după cum urmează: formăm un graf complet $G' = (V, E')$ unde $E' = \{(i, j) : i, j \in V\}$ şi definim funcţia de cost c prin:

$$c(i, j) = \begin{cases} 0 & \text{dacă } (i, j) \in E, \\ 1 & \text{dacă } (i, j) \notin E. \end{cases}$$

Instanţa problemei PCV este $(G', c, 0)$ şi poate fi foarte uşor determinată în timp polinomial.

Arătăm acum că graful G are un ciclu hamiltonian dacă, şi numai dacă, graful G' conţine un ciclu de cost cel mult 0. Să presupunem că graful G are un ciclu hamiltonian h. Fiecare muchie din $h \in E$, deci are cost 0 în G'. Deci h este un ciclu în G' cu costul 0. Invers, să presupunem că graful G' are un ciclu h' de cost cel mult 0. Deoarece costurile muchiilor din E' sunt 0 sau 1, costul ciclului este exact 0. În concluzie, h' conţine numai muchii din E, deci h este un ciclu hamiltonian în graful G. ■

Exerciţii

36.5-1 Problema izomorfismului subgrafului cere ca, dându-se două grafuri G_1 şi G_2, să se verifice dacă G_1 este izomorf cu un subgraf al lui G_2. Arătaţi că această problemă este NP-completă.

36.5-2 Dându-se o matrice A de dimensiuni $m \times n$ cu elemente numere întregi şi un vector b cu elemente numere întregi de dimensiune m, ***problema de programare a întregilor 0-1*** cere determinarea existenţei unui vector x de dimensiune n având elemente din mulţimea $\{0, 1\}$ astfel încât $Ax \leq b$. Demonstraţi că această problemă este NP-completă. (*Indicaţie:* Reduceţi de la 3-FNC-SAT.)

36.5-3 Arătaţi că problema sumei submulţimii este rezolvabilă în timp polinomial dacă valoarea ţintă t este reprezentată unar.

36.5-4 Problema partiţionării mulţimii are ca intrare o mulţime S de numere întregi. Determinaţi dacă numerele pot fi partiţionate în două mulţimi A şi $\overline{A} = S - A$ astfel încât $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$. Arătaţi că problema partiţionării mulţimii este NP-completă.

36.5-5 Arătaţi că problema drumului hamiltonian este NP-completă.

36.5-6 Problema celui mai lung ciclu elementar este problema determinării unui ciclu elementar (fără vârfuri care se repetă) de lugime maximă într-un graf. Arătaţi că această problemă este NP-completă.

36.5-7 Profesorul Marconi afirmă că subgraful folosit ca graf ajutător A în demonstrarea teoremei 36.14 este mai complicat decât este necesar: vârfurile z_3 şi z_4 din figura 36.15(a) şi vârfurile de deasupra şi dedesubtul lor nu sunt necesare. Are profesorul dreptate? Adică, funcţionează reducerea cu această versiune mai mică a grafului ajutător, sau dispare proprietatea de "sau/sau" a muchiilor?

Probleme

36-1 Mulţime independentă

O ***mulţime independentă*** a unui graf $G = (V, E)$ este o submulţime $V' \subseteq V$ de vârfuri astfel încât fiecare muchie din E este incidentă la cel mult un vârf din V'. ***Problema mulţimii independente*** cere găsirea unei mulţimi extern stabile de dimensiune maximă în G.

a. Formulaţi o problemă de decizie corespunzătoare pentru problema mulţimii independente şi demonstraţi că este NP-completă. (*Indicaţie:* Reduceţi de la problema clicii.)

b. Să presupunem că avem o subrutină care rezolvă problema de decizie definită la punctul (a). Găsiţi un algoritm pentru găsirea unei mulţimi independente de dimensiune maximă. Timpul de execuţie al algoritmului ar trebui să fie polinomial în $|V|$ şi $|E|$. (Apelul subrutinei este considerat a fi un singur pas.)

Cu toate că problema de decizie a mulţimii independente este NP-completă, câteva cazuri particulare pot fi rezolvate în timp polinomial.

c. Găsiţi un algoritm eficient pentru rezolvarea problemei extern stabile când fiecare vârf al lui G are gradul 2. Analizaţi timpul de execuţie şi demonstraţi că algoritmul propus este corect.

d. Găsiţi un algoritm eficient pentru rezolvarea problemei mulţimii extern stabile când un graf G este bipartit. Analizaţi timpul de excuţie şi demonstraţi că algoritmul propus este corect. (*Indicaţie:* Folosiţi rezultatele de la secţiunea 27.3)

36-2 Colorarea grafului

O ***k-colorare*** a unui graf neorientat $G = (V, E)$ este o funcţie $c : V \to \{1, 2, \ldots, k\}$ astfel încât $c(u) \neq c(v)$ pentru fiecare muchie $(u, v) \in E$. Cu alte cuvinte, numerele $1, 2, \ldots, k$ reprezintă k culori diferite. ***Problema colorării grafului*** cere determinarea numărului minim de culori necesare pentru colorarea unui graf dat.

a. Găsiţi un algoritm eficient pentru determinarea unei 2-colorări a unui graf, dacă există una.

b. Reformulaţi problema colorării grafului într-o problemă de decizie. Arătaţi că problema de decizie astfel formulată este rezolvabilă în timp polinomial dacă, şi numai dacă, problema colorării grafului este rezolvabilă în timp polinomial.

c. Fie limbajul 3-COL definit ca mulţimea grafurilor care pot fi 3-colorate. Arătaţi că, dacă 3-COL este NP-completă, atunci problema de decizie formulată la punctul (b) este NP-completă.

Pentru a demonstra că 3-COL este NP-completă, folosiţi o reducere de la 3-FNC-SAT. Dându-se o formulă ϕ cu m propoziţii şi n variabile $x_1, x_2, \ldots, x_n$, construim un graf $G = (V, E)$ după cum urmează. Mulţimea V constă dintr-un vârf pentru fiecare variabilă, un vârf pentru negaţia fiecărei variabile, cinci vârfuri pentru fiecare propoziţie şi trei vârfuri speciale: ADEVĂRAT, FALS şi ROŞU. Muchiile literalilor formează un triunghi pe vârfurile speciale şi, de asemenea, formează un triunghi pe $x_i, \neg x_i$ şi ROŞU pentru $i = 1, 2, \ldots, n$.

d. Argumentaţi că în orice 3-colorare c a unui graf care conţine muchiile literalilor, o variabilă şi negaţia ei sunt colorate cu c(ADEVĂRAT), respectiv c(FALS), sau invers. Demonstraţi că pentru orice atribuire de adevăr pentru ϕ, există o 3-colorare a grafului care conţine numai literalii muchiilor.

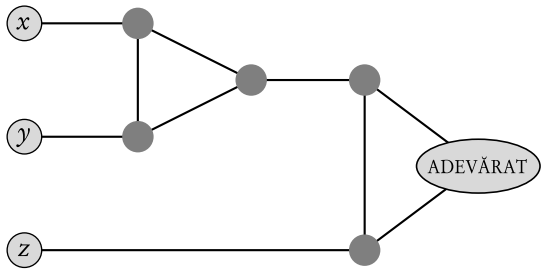


Figura 36.20 Graful ajutător corespunzător propoziţiei $(x \vee y \vee z)$ folosit în problema 36-2.

Graful ajutător, prezentat în figura 36.20, este folosit pentru a îndeplini condiţia corespunzătoare unei clauze $(x \vee y \vee z)$. Fiecare propoziţie necesită o copie unică a celor cinci vârfuri care sunt haşurate cu negru în figură. Ele sunt conectate, aşa cum se arată, cu literalii propoziţiei şi cu vârful special ADEVĂRAT.

e. Demonstraţi că, dacă x, y şi z sunt coloraţi cu c(ADEVĂRAT) sau c(FALS) atunci graful ajutător este 3-colorabil dacă şi numai dacă cel puţin unul dintre vârfurile x, y şi z este colorat cu c(ADEVĂRAT).

f. Completaţi demonstraţia faptului că 3-COL este NP-completă.

Note bibliografice

Garey şi Johnson [79] furnizează un ghid pentru NP-completitudine foarte bine realizat, discutând pe larg teoria şi prezentând un catalog cu multe probleme despre care se ştia în 1979 că sunt NP-complete. (Lista domeniilor în care apar probleme NP-complete este preluată din cuprinsul lucrării lor.) Hopcroft, Ullman [104] şi Lewis, Papadimitriou [139] au o tratare bună a NP-completitudinii în contextul teoriei complexităţii. Aho, Hopcroft, Ullman [4] acoperă şi ei NP-completitudinea şi dau câteva reduceri, incluzându-le pe cele de la problema acoperirii cu vârfuri şi pe cea a grafului hamiltonian.

Clasa P a fost introdusă în 1964 de Cobham [44] şi, independent, de Edmonds [61] în 1965. Acesta din urmă a introdus şi clasa NP şi a formulat ipoteza P $\neq$ NP. Noţiunea de NP-completitudine a fost propusă în 1971 de Cook [49], cel care a dat primele demonstraţii ale NP-completitudinii pentru problema satisfiabilităţii formulei şi problema satisfiabilităţii formulei 3-FNC. Levin [138] a descoperit, independent, noţiunea, dând o dovadă a NP-completitudinii pentru o problemă de acoperire. Karp [116] a introdus metodologia reducerii în 1972 şi a demonstrat marea varietate a problemelor NP-complete. Lucrarea lui Karp include demonstraţia originală a NP-completitudinii problemelor clicii, a acoperirii cu vârfuri şi a ciclului hamiltonian. De atunci, mulţi cercetători au dovedit NP-completitudinea mai multor sute de probleme.

Demonstraţia teoremei 36.14 a fost adaptată din lucrarea aparţinând lui Papadimitriou şi Steiglitz [154].

37 Algoritmi de aproximare

Multe probleme de importanţă practică sunt NP-complete, dar sunt prea importante pentru a fi abandonate pentru simpluls fapt că obţinerea unei soluţii optimale este netratabilă. Dacă o problemă este NP-completă, este puţin probabil să găsim un algoritm în timp polinomial care să rezolve problema exact, dar aceasta nu înseamnă că am pierdut orice speranţă. Există două abordări pentru ocolirea NP-completitudinii. Mai întâi, dacă datele de intrare sunt de dimensiuni reduse, un algoritm cu timp de execuţie exponenţial poate fi absolut satisfăcător. Apoi, ar putea fi, încă, posibil să obţinem soluţii *aproape optimale* în timp polinomial (fie în cazul cel mai defavorabil, fie în cazul mediu). În practică, aproape-optimalitatea este, deseori, suficient de bună. Un algoritm care returnează soluţii aproape optimale se numeşte ***algoritm de aproximare***. Acest capitol prezintă algoritmi de aproximare în timp polinomial pentru câteva probleme NP-complete.

Limite de performanţă pentru algoritmi de aproximare

Să presupunem că lucrăm la o problemă de optimizare în care fiecare soluţie potenţială are un cost pozitiv şi că dorim să găsim o soluţie aproape optimală. În funcţie de problemă, o soluţie optimală ar putea fi definită ca fiind una cu cost posibil maxim sau una cu cost posibil minim; problema poate fi o problemă de maximizare sau de minimizare.

Spunem că un algoritm de aproximare pentru problemă are ***marginea raportului*** de $\rho(n)$ dacă, pentru orice date de intrare de dimensiune n, costul C al soluţiei produse de algoritmul de aproximare se abate cu un factor de $\rho(n)$ de costul $C^\star$ al unei soluţii optimale:

$$\max\left(\frac{C}{C^\star}, \frac{C^\star}{C}\right) \leq \rho(n) \tag{37.1}$$

Această definiţie este valabilă atât pentru problemele de minimizare cât şi de maximizare. Pentru o problemă de maximizare, $0 < C \leq C^\star$, şi raportul $C^\star/C$ produce factorul cu care costul soluţiei optimale este mai mare decât costul soluţiei aproximative. Similar, pentru o problemă de minimizare, $0 < C^\star \leq C$, şi raportul $C/C^\star$ produce factorul cu care costul soluţiei aproximative este mai mare decât costul soluţiei optimale. Deoarece toate soluţiile se presupun ca având cost pozitiv, aceste rapoarte sunt totdeauna corect definite. Marginea raportului unui algoritm de aproximare nu este niciodată mai mică decât 1, deoarece $C/C^\star < 1$ implică $C^\star/C > 1$. Un algoritm optimal are marginea raportului egală cu 1, iar un algoritm de aproximare cu o margine mare a raportului poate returna o soluţie care este mult mai rea decât cea optimală.

Uneori, este mai convenabil să lucrăm cu o măsură a erorii relative. Pentru orice date de intrare, ***eroarea relativă*** a algoritmului de aproximare este definită astfel:

$$\frac{|C - C^\star|}{C^\star},$$

unde, ca mai înainte, $C^\star$ este costul unei soluţii optimale, şi C este costul soluţiei produse de algoritmul de aproximare. Eroarea relativă este întotdeauna nenegativă. Un algoritm de aproximare are o ***margine a erorii relative*** de $\epsilon(n)$ dacă

$$\frac{|C - C^\star|}{C^\star} \leq \epsilon(n). \tag{37.2}$$

Din definiţii, rezultă că marginea erorii relative poate fi mărginită, ca funcţie de marginea raportului:

$$\epsilon(n) \leq \rho(n) - 1. \tag{37.3}$$

(Pentru o problemă de minimizare, aceasta este o egalitate, în timp ce pentru o problemă de maximizare avem $\epsilon(n) = (\rho(n) - 1)/\rho(n)$, care satisface inegalitatea (37.3) deoarece $\rho(n) \geq 1$.)

Pentru multe probleme au fost dezvoltaţi algoritmi de aproximare cu o margine a raportului fixă, independentă de n. Pentru astfel de probleme, utilizăm notaţia mai simplă ρ sau ϵ, indicând independenţa de n.

Pentru unele probleme, nu s-au putut realiza algoritmi de aproximare în timp polinomial cu marginea raportului fixă. Pentru astfel de probleme, cel mai bun lucru pe care îl putem face este să permitem creşterea marginii raportului ca o funcţie de dimensiunea n a datelor de intrare. Un exemplu de astfel de probleme este problema de acoperire a mulţimii, prezentată în secţiunea 37.3.

Unele probleme NP-complete permit construirea unor algoritmi de aproximare care pot atinge margini ale raportului din ce în ce mai mici (sau, echivalent, margini ale erorii relative din ce în ce mai mici) prin utilizarea unui timp de calcul din ce în ce mai mare. Adică, există un echilibru între timpul de calcul şi calitatea aproximării. Un exemplu este problema sumei submulţimii, studiată în secţiunea 37.4. Această situaţie este suficient de importantă pentru a merita un nume al ei.

O ***schemă de aproximare*** pentru o problemă de optimizare este un algoritm de aproximare care are ca date de intrare nu doar o instanţă a problemei, ci şi o valoare $\epsilon > 0$, astfel încât, pentru orice ϵ fixat, schema este un algoritm de aproximare cu marginea erorii relative egală cu ϵ. Spunem că o schemă de aproximare este o ***schemă de aproximare în timp polinomial*** dacă, pentru orice $\epsilon > 0$ fixat, schema se execută într-un timp polinomial în dimensiunea n a instanţei sale de intrare.

Timpul de execuţie a unei scheme de aproximare în timp polinomial nu ar trebui să crească prea repede pe măsură ce ϵ scade. Ideal ar fi ca dacă ϵ scade cu un factor constant, timpul de execuţie pentru a atinge aproximarea dorită să nu crească cu mai mult decât un factor constant. Cu alte cuvinte, am dori ca timpul de execuţie să fie polinomial atât în $1/\epsilon$ cât şi în n.

Spunem că o schemă de aproximare este o ***schemă de aproximare în timp complet polinomial*** dacă timpul său de execuţie este polinomial atât în $1/\epsilon$ cât şi în dimensiunea n a instanţei de intrare, unde ϵ este marginea erorii relative a schemei. De exemplu, schema ar putea avea un timp de execuţie de $(1/\epsilon)^2 n^3$. Cu o astfel de schemă, orice scădere cu un factor constant a lui ϵ se poate obţine cu o creştere cu un factor constant corespunzătoare a timpului de execuţie.

Rezumatul capitolului

Primele trei secţiuni ale acestui capitol prezintă exemple de algoritmi de aproximare în timp polinomial pentru probleme NP-complete, iar ultima secţiune prezintă o schemă de aproximare în timp complet polinomial. Secţiunea 37.1 începe cu un studiu al problemei acoperirii cu vârfuri, o problemă NP-completă de minimizare care are un algoritm de aproximare cu marginea raportului egală cu 2. Secţiunea 37.2 prezintă un algoritm de aproximare cu marginea raportului egală cu 2, pentru cazul problemei comis-voiajorului, în care funcţia de cost satisface inegalitatea triunghiului. De asemenea, se arată că, fără inegalitatea triunghiului, un algoritm de ϵ-aproximare

nu poate exista decât dacă P = NP. În secţiunea 37.3, arătăm cum poate fi folosită o metodă greedy ca algoritm de aproximare eficient pentru problema acoperirii mulţimii, obţinând o acoperire al cărei cost este în cel mai defavorabil caz cu un factor logaritmic mai mare decât costul optimal. Finalmente, secţiunea 37.4 prezintă o schemă de aproximare în timp complet polinomial pentru problema sumei submulţimii.

37.1. Problema acoperirii cu vârfuri

Problema acoperirii cu vârfuri a fost definită şi i s-a demonstrat NP-completitudinea în secţiunea 36.5.2. O ***acoperire cu vârfuri*** a unui graf neorientat $G = (V, E)$ este o submulţime $V' \subseteq V$, astfel încât, dacă (u, v) este o muchie din G, atunci fie $u \in V'$, fie $v \in V'$ (fie ambele). Dimensiunea unei acoperiri cu vârfuri este dată de numărul vârfurilor pe care le conţine.

Problema acoperirii cu vârfuri constă în a găsi o acoperire cu vârfuri de dimensiune minimă pentru un graf neorientat dat. Acoperirea cu vârfuri având această proprietate o numim ***acoperire cu vârfuri optimală***. Această problemă este NP-dificilă, deoarece problema de decizie asociată este NP-completă, conform teoremei 36.12.

Chiar dacă o acoperire cu vârfuri optimală pentru un graf G poate fi dificil de găsit, o acoperire cu vârfuri aproape optimală nu este prea dificil de găsit. Următorul algoritm de aproximare primeşte ca intrare un graf neorientat G şi returnează o acoperire cu vârfuri a cărei dimensiune este, cu siguranţă, nu mai mult de două ori mai mare decât dimensiunea unei acoperiri cu vârfuri optimale.

ACOPERIRE-CU-VÂRFURI-APROX(G)
1: $C \leftarrow \emptyset$
2: $E' \leftarrow E[G]$
3: **cât timp** $E' \neq \emptyset$ **execută**
4: fie (u, v) o muchie arbitrară din E'
5: $C \leftarrow C \cup \{u, v\}$
6: şterge din E' fiecare muchie incidentă cu u sau cu v
7: **returnează** C

Figura 37.1 ilustrează modul de funcţionare al algoritmului ACOPERIRE-CU-VÂRFURI-APROX. Variabila C conţine acoperirea cu vârfuri în curs de construire. Linia 1 iniţializează variabila C cu mulţimea vidă. Linia 2 atribuie variabilei E' o copie a mulţimii de vârfuri $E[G]$ a grafului. Bucla din liniile 3–6 alege, în mod repetat, o muchie (u, v) din E', adaugă capetele sale u şi v la C şi şterge toate muchiile din E' care sunt acoperite fie de u, fie de v. Timpul de execuţie al acestui algoritm este $O(E)$ şi utilizează o structură de date corespunzătoare pentru reprezentarea lui E'.

Teorema 37.1 ACOPERIRE-CU-VÂRFURI-APROX are marginea raportului de 2.[1]

Demonstraţie. Mulţimea de vârfuri C returnată de ACOPERIRE-CU-VÂRFURI-APROX este o acoperire cu vârfuri, deoarece algoritmul ciclează până când fiecare muchie din $E[G]$ este acoperită de un vârf oarecare din C.

[1] Soluţia furnizată de algoritm conţine de cel mult două ori mai multe vârfuri decât soluţia optimă. – n.t.

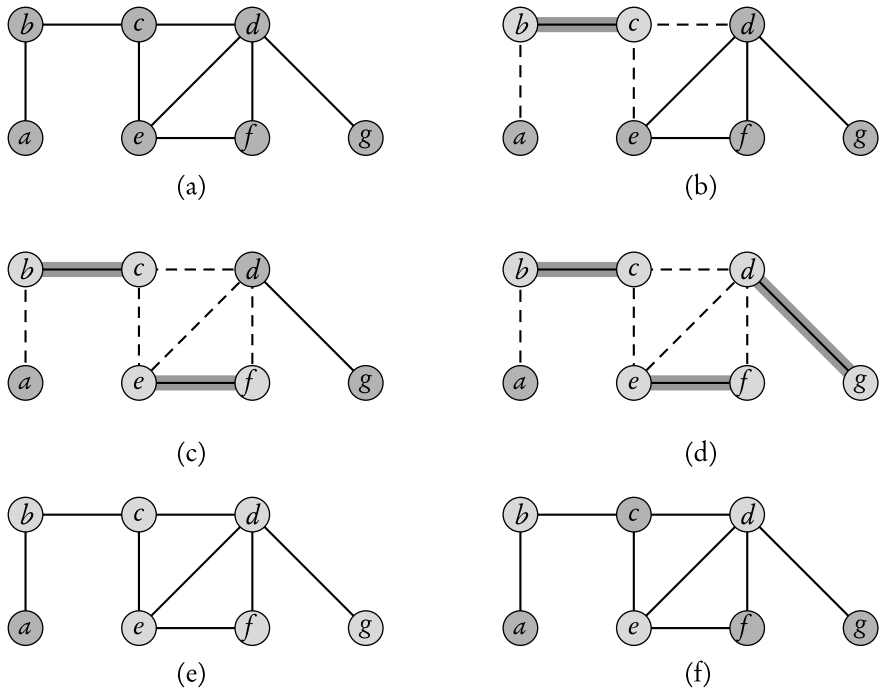


Figura 37.1 Modul de operare al algoritmului ACOPERIRE-CU-VÂRFURI-APROX. **(a)** Graful de intrare G, care are 7 vârfuri şi 8 muchii. **(b)** Muchia (b, c), indicată îngroşat, este prima muchie aleasă de ACOPERIRE-CU-VÂRFURI-APROX. Vârfurile b şi c, indicate cu haşură, sunt adăugate acoperirii cu vârfuri C în curs de creare. Muchiile (a, b), (c, e) şi (c, d), indicate cu linie întreruptă, sunt şterse deoarece sunt acoperite de unele din vârfurile din C. **(c)** Muchia (e, f) este adăugată la C. **(d)** Muchia (d, g) este adăugată la C. **(e)** Mulţimea C, care este acoperirea cu vârfuri produsă de ACOPERIRE-CU-VÂRFURI-APROX, conţine cele şase vârfuri b, c, d, e, f, g. **(f)** Acoperirea cu vârfuri optimală pentru această problemă conţine doar trei vârfuri: b, d şi e.

Pentru a vedea că ACOPERIRE-CU-VÂRFURI-APROX returnează o acoperire cu vârfuri care este cel mult de două ori mai mare decât o acoperire optimală, să notăm cu A mulţimea vârfurilor alese în linia 4 a algoritmului. Nu există în A două muchii care să aibă un capăt comun, deoarece, o dată ce în linia 4 am ales o muchie, toate celelalte muchii care sunt incidente cu unul din capetele muchiei alese sunt şterse din E' la linia 6. Prin urmare, fiecare execuţie a liniei 5 adaugă două noi vârfuri la mulţimea C, şi $|C| = 2|A|$. Pentru a acoperi muchiile mulţimii A, orice acoperire cu vârfuri – în particular o acoperire optimală $C^\star$ – trebuie să includă cel puţin unul din capetele fiecărei muchii din A. Deoarece în A nu există două muchii care să partajeze acelaşi capăt, nici un vârf din mulţime nu este incident cu mai mult de o muchie din A. Prin urmare $|A| \leq |C^\star|$ şi $|C| \leq 2|C^\star|$, ceea ce demonstrează teorema. ∎

Exerciţii

37.1-1 Daţi un exemplu de graf pentru care ACOPERIRE-CU-VÂRFURI-APROX generează întotdeauna o soluţie suboptimală.

37.1-2 Profesorul Nixon propune următoarea euristică pentru rezolvarea problemei acoperirii cu vârfuri. Se selectează, în mod repetat, un vârf având gradul cel mai mare şi se şterg toate muchiile sale incidente. Daţi un exemplu pentru a arăta că euristica profesorului nu are marginea raportului egală cu 2.

37.1-3 Daţi un algoritm greedy eficient care găseşte în timp liniar o acoperire cu vârfuri optimală pentru un arbore.

37.1-4 Din demonstraţia teoremei 36.12, ştim că problema acoperirii cu vârfuri şi problema NP-completă a clicii sunt complementare în sensul că o acoperire cu vârfuri este complementul unei clici de dimensiune maximă în graful complementar. Implică această relaţie faptul că există un algoritm de aproximare pentru problema clicii cu marginea raportului constantă? Justificaţi răspunsul.

37.2. Problema comis-voiajorului

În problema comis-voiajorului, introdusă în secţiunea 36.5.5, se dă un graf neorientat complet, $G = (V, E)$, care are un cost întreg nenegativ $c(u, v)$ asociat fiecărei muchii $(u, v) \in E$. Trebuie să găsim un ciclu hamiltonian (un tur) de cost minim al lui G. Ca o extensie a notaţiei noastre, să notăm cu $c(A)$ costul total al muchiilor submulţimii $A \subseteq E$:

$$c(A) = \sum_{(u,v) \in A} c(u, v).$$

În multe situaţii practice, este mai convenabil să ne deplasăm direct dintr-un loc u într-un loc w; deplasarea printr-un punct intermediar v nu poate fi mai puţin costisitoare. Cu alte cuvinte, reducerea unui punct intermediar nu duce niciodată la creşterea costului. Pentru a formaliza această noţiune, vom spune că funcţia de cost c satisface ***inegalitatea triunghiului*** dacă, pentru toate vârfurile $u, v, w \in V$, are loc

$$c(u, w) \leq c(u, v) + c(v, w).$$

Inegalitatea triunghiului este naturală şi, în multe aplicaţii, este satisfăcută automat. De exemplu, dacă vârfurile grafului sunt puncte din plan şi costul deplasării între două puncte este distanţa euclidiană uzuală dintre acestea, atunci inegalitatea triunghiului este satisfăcută.

Aşa cum arată exerciţiul 37.2-1, introducerea unei restricţii asupra funcţiei de cost, astfel încât aceasta să satisfacă inegalitatea triunghiului, nu alterează NP-completitudinea problemei comis-voiajorului. Astfel, este puţin probabil să putem găsi un algoritm în timp polinomial pentru rezolvarea cu exactitate a acestei probleme. Prin urmare vom căuta algoritmi buni de aproximare.

În secţiunea 37.2.1 examinăm un algoritm de aproximare pentru problema comis-voiajorului cu inegalitatea triunghiului şi care are o margine a raportului egală cu 2. În secţiunea 37.2.2 arătăm că fără inegalitatea triunghiului un algoritm de aproximare cu margine constantă a raportului există numai dacă P = NP.

37.2.1. Problema comis-voiajorului cu inegalitatea triunghiului

Următorul algoritm determină un tur aproape optimal al unui graf neorientat G folosind algoritmul arborelui de acoperire minimă AAM-PRIM din secţiunea 24.2. Vom vedea că, în situaţia în care funcţia de cost satisface inegalitatea triunghiului, turul returnat de acest algoritm nu este mai lung decât dublul lungimii drumului optimal.

PCV-TUR-APROX(G, c)
1: selectează un vârf $r \in V[G]$ ca vârf "rădăcină"
2: folosind algoritmul AAM-PRIM(G, c, r) determină un arbore de acoperire minimă T pentru G, având rădăcina r
3: fie L lista vârfurilor lui T parcurse în preordine
4: **returnează** ciclul hamiltonian H care conţine vârfurile în ordinea din L

Să ne reamintim, din secţiunea 13.1, că o parcurgere în preordine vizitează în mod recursiv fiecare vârf al arborelui şi listează un vârf în momentul în care a fost întâlnit prima dată, înainte ca oricare dintre fiii săi să fie vizitat.

Figura 37.2 ilustrează modul de operare al algoritmului PCV-TUR-APROX. Partea (a) a figurii arată mulţimea de vârfuri dată şi partea (b) arată arborele de acoperire minimă T având vârful rădăcină a, produs de algoritmul AAM-PRIM. Partea (c) arată modul în care vârfurile sunt vizitate de o parcurgere în preordine a lui T şi partea (d) prezintă turul returnat de PCV-TUR-APROX. Partea (e) prezintă un tur optimal, care este cu aproximativ 23% mai scurt.

Timpul de execuţie al PCV-TUR-APROX este $\Theta(E) = \Theta(V^2)$, deoarece graful de intrare este un graf complet (vezi exerciţiul 24.2-2). Vom arăta, acum, că, dacă funcţia de cost pentru o instanţă a problemei comis-voiajorului satisface inegalitatea triunghiului, atunci PCV-TUR-APROX returnează un tur al cărui cost nu este mai mare decât dublul costului unui tur optimal.

Teorema 37.2 PCV-TUR-APROX este un algoritm de aproximare pentru problema comis-voiajorului cu inegalitatea triunghiului şi are o margine a raportului egală cu 2.

Demonstraţie. Fie $H^\star$ un tur optimal pentru mulţimea de vârfuri dată. Un enunţ echivalent cu enunţul teoremei este că $c(H) \leq 2c(H^\star)$, unde H este turul returnat de PCV-TUR-APROX. Deoarece prin ştergerea oricărei muchii dintr-un tur obţinem un arbore de acoperire, dacă T este un arbore de acoperire minim pentru mulţimea de vârfuri dată, atunci

$$c(T) \leq C(H^\star) \tag{37.4}$$

Un ***drum complet*** al lui T listează vârfurile în momentul în care acestea sunt vizitate prima dată şi oricând se revine la ele după vizitarea unui subarbore. Să notăm acest drum cu W. Drumul complet, pentru exemplul nostru produce ordinea

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a.$$

Deoarece drumul complet traversează fiecare muchie a lui T exact de două ori, avem

$$c(W) = 2c(T). \tag{37.5}$$

Ecuaţiile (37.4) şi (37.5) implică

$$c(W) \leq 2c(H^\star) \tag{37.6}$$

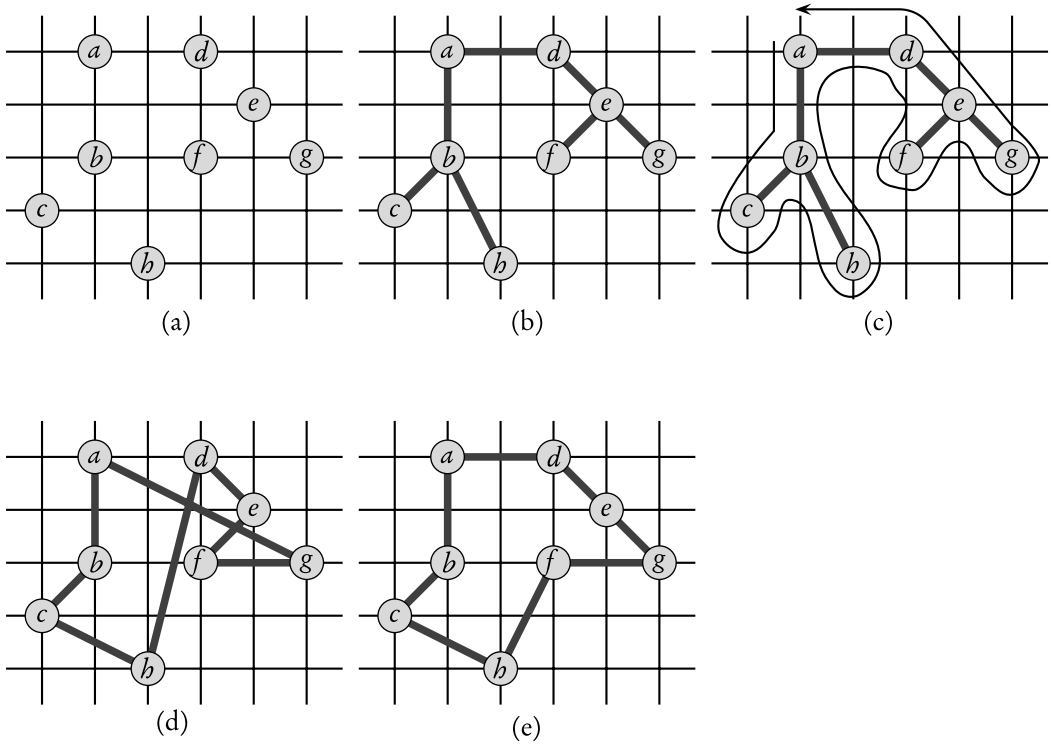


Figura 37.2 Modul de operare al algoritmului PCV-Tur-Aprox. **(a)** Mulţimea de puncte dată, situată pe o grilă având coordonatele întregi. De exemplu, f este o unitate la dreapta şi două unităţi mai sus decât h. Ca funcţie de cost între două puncte, este utilizată distanţa euclidiană uzuală. **(b)** Un arbore de acoperire minimă T a acestor puncte, aşa cum este determinat de AAM-Prim. Vârful a este vârf rădăcină. Vârfurile sunt etichetate întâmplător, astfel încât ele sunt adăugate în ordine alfabetică arborelui principal construit de AAM-Prim. **(c)** Un drum prin T, care începe în a. Un drum complet al arborelui vizitează nodurile în ordinea a, b, c, b, h, b, a, d, e, f, e, g, e, d, a. O parcurgere în preordine a lui T listează un vârf doar prima dată când este întâlnit şi produce ordinea a, b, c, h, d, e, f, g. **(d)** Un tur al vârfurilor, obţinut prin vizitarea vârfurilor în ordinea dată de parcurgerea în preordine. Acesta este turul H returnat de PCV-Tur-Aprox. Costul său total este aproximativ 19.074. **(e)** Un tur optimal $H^\star$ pentru mulţimea de vârfuri dată. Costul său total este aproximativ 14.715.

şi, deci, costul lui W este mai mic decât dublul costului unui tur optimal.

Din păcate, W, în general, nu este un tur, deoarece vizitează unele vârfuri de mai multe ori. Totuşi, pe baza inegalităţii triunghiului, putem şterge din W o vizită a oricărui vârf, fără ca prin aceasta costul să crească. (Dacă un vârf v este şters din W între vizitele vârfurilor u şi w, ordinea rezultată precizează deplasarea directă de la u la w.) Prin aplicarea acestei operaţii în mod repetat, putem şterge din W toate apariţiile vârfurilor, cu excepţia primelor apariţii. În exemplul nostru aceasta produce ordinea

$$a, b, c, h, d, e, f, g.$$

Această ordine este aceeaşi cu cea obţinută printr-o parcurgere în preordine a arborelui T. Fie H

ciclul corespunzător acestei parcurgeri în preordine. Acesta este un ciclu hamiltonian, deoarece fiecare vârf este vizitat exact o singură dată. De fapt este ciclul determinat de PCV-TUR-APROX. Deoarece H este obţinut prin ştergerea vârfurilor din drumul complet W, avem

$$c(H) \leq c(W). \tag{37.7}$$

Combinarea inegalităţilor (37.6) şi (37.7) încheie demonstraţia. ∎

În pofida marginii bune a raportului, asigurată de teorema 37.2, algoritmul PCV-TUR-APROX nu reprezintă de obicei o soluţie practică pentru această problemă. Există şi alţi algoritmi de aproximare care de obicei se comportă mult mai bine în practică (vezi referinţele de la sfârşitul capitolului).

37.2.2. Problema generală a comis-voiajorului

Dacă renunţăm la presupunerea că funcţia de cost c satisface inegalitatea triunghiului, nu mai pot fi găsite tururi aproximative bune în timp polinomial decât dacă P = NP.

Teorema 37.3 Dacă P ≠ NP şi $\rho \geq 1$, atunci, pentru problema generală a comis-voiajorului, nu există un algoritm de aproximare în timp polinomial cu marginea raportului egală cu ρ.

Demonstraţie. Demonstraţia se face prin reducere la absurd. Să presupunem contrariul concluziei, şi anume că există un anumit număr $\rho \geq 1$ astfel încât există un algoritm de aproximare A în timp polinomial, care are marginea raportului egală cu ρ. Fără a pierde generalitatea, presupunem că ρ este număr întreg. Dacă este necesar, îl rotunjim în sus. Vom arăta cum să folosim A pentru a rezolva în timp polinomial instanţe ale problemei ciclului hamiltonian (definită în secţiunea 36.5.5). Conform teoremei 36.14, problema ciclului hamiltonian este NP-completă. Ca urmare, conform teoremei 36.4, rezolvarea ei în timp polinomial implică faptul că P = NP.

Fie $G = (V, E)$ o instanţă a problemei ciclului hamiltonian. Dorim să determinăm, în mod eficient, dacă G conţine un ciclu hamiltonian. Pentru aceasta, vom utiliza algoritmul de aproximare A. Transformăm G într-o instanţă a problemei comis-voiajorului, după cum urmează. Fie $G' = (V, E')$ graful complet al lui V, adică

$$E' = \{(u, v) | u, v \in V, u \neq v\}.$$

Atribuim câte un cost întreg fiecărei muchii din E' după cum urmează:

$$c(u, v) = \begin{cases} 1 & \text{dacă } (u, v) \in E, \\ \rho|V| + 1 & \text{altfel.} \end{cases}$$

Folosind o reprezentare a lui G, reprezentări ale lui G' şi c pot fi create în timp polinomial în $|V|$ şi $|E|$.

Acum, să considerăm problema comis-voiajorului (G', c). Dacă graful original G are un ciclu hamiltonian H, atunci funcţia de cost c atribuie fiecărei muchii a lui H un cost egal cu 1, şi, astfel, (G', c) conţine un tur de cost $|V|$. Pe de altă parte, dacă G nu conţine un ciclu hamiltonian, atunci orice tur al lui G' trebuie să folosească unele din muchiile care nu sunt în E. Dar orice tur care foloseşte o muchie din afara lui E are un cost cel puţin egal cu

$$(\rho|V| + 1) + (|V| - 1) > \rho|V|.$$

Deoarece muchiile din afara lui G sunt atât de costisitoare, există o distanţă mare între costul unui tur care este ciclu hamiltonian în G (cost $|V|$) şi costul oricărui alt tur (cost superior lui $\rho|V|$).

Ce se întâmplă dacă aplicăm algoritmul de aproximare A problemei comis-voiajorului (G', c)? Deoarece A garantează returnarea unui tur de cost nu mai mult decât de ρ ori superior costului unui tur optimal, dacă G conţine un ciclu hamiltonian, atunci A trebuie să îl returneze. Dacă G nu are ciclu hamiltonian, atunci A returnează un tur de cost superior lui $\rho|V|$. Prin urmare, putem folosi algoritmul A pentru rezolvarea problemei ciclului hamiltonian în timp polinomial. ■

Exerciţii

37.2-1 Arătaţi cum se poate transforma în timp polinomial o instanţă a problemei comis-voiajorului într-o altă instanţă a cărei funcţie de cost satisface inegalitatea triunghiului. Cele două instanţe trebuie să aibă aceeaşi mulţime de tururi optimale. Presupunând că P $\neq$ NP, explicaţi de ce o astfel de transformare în timp polinomial nu contrazice teorema 37.3.

37.2-2 Consideraţi următoarea ***euristică a celui mai apropiat punct*** pentru construirea unui tur aproximativ al comis-voiajorului. Începeţi cu un ciclu trivial care constă dintr-un singur vârf ales arbitrar. La fiecare pas, identificaţi vârful u care nu aparţine ciclului, dar a cărui distanţă faţă de orice vârf din ciclu este minimă. Modificaţi ciclul astfel încât el să conţină vârful u (inseraţi vârful u imediat după vârful v). Să notăm cu v vârful din ciclu care este cel mai apropiat de u. Repetaţi procedeul până când toate vârfurile aparţin ciclului. Demonstraţi că această euristică returnează un tur al cărui cost total nu este mai mare decât dublul costului unui tur optimal.

37.2-3 Problema comis-voiajorului strâmtorat constă în aflarea unui ciclu hamiltonian, astfel încât lungimea celei mai mari muchii din ciclu să fie minimizată. Presupunând că funcţia de cost satisface inegalitatea triunghiului, arătaţi că, pentru această problemă, există un algoritm de aproximare în timp polinomial cu marginea raportului egală cu 3. (*Indicaţie:* Arătaţi, în mod recursiv, că putem vizita exact o dată toate nodurile dintr-un arbore de acoperire parcurgând un drum complet în arbore şi sărind peste unele noduri, dar fără să se sară peste mai mult de două noduri intermediare consecutive.)

37.2-4 Să presupunem că vârfurile unei instanţe a problemei comis-voiajorului sunt puncte din plan şi că funcţia de cost $c(u, v)$ este distanţa euclidiană între punctele u şi v. Arătaţi că un tur optimal nu se autointersectează.

37.3. Problema acoperirii mulţimii

Problema acoperirii mulţimii este o problemă de optimizare care modelează multe probleme de selecţie a resurselor şi care generalizează problema NP-completă a acoperirii cu vârfuri şi este, prin urmare, şi NP-dificilă. Totuşi, algoritmul de aproximare creat pentru problema acoperirii cu vârfuri nu se aplică şi aici, şi, prin urmare, avem nevoie să încercăm alte abordări. Vom examina o euristică greedy simplă cu o margine logaritmică a raportului. Adică, pe măsură ce dimensiunea instanţei devine mai mare, dimensiunea soluţiei aproximative poate creşte, relativ

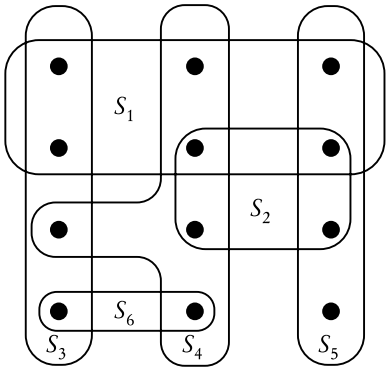


Figura 37.3 O instanţă $(X, \mathcal{F})$ a problemei acoperirii mulţimii, unde X constă din cele 12 puncte negre şi $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. O mulţime de acoperire de dimensiune minimă este $\mathcal{C} = \{S_3, S_4, S_5\}$. Algoritmul greedy produce o acoperire de dimensiune 4 prin selectarea, în ordine, a mulţimilor S_1, S_4, S_5 şi S_3.

la dimensiunea unei soluţii optimale. Deoarece funcţia logaritm creşte destul de încet, algoritmul de aproximare poate produce rezultate utile.

O instanţă $(X, \mathcal{F})$ a ***problemei acoperirii mulţimii*** constă dintr-o mulţime finită X şi o familie $\mathcal{F}$ de submulţimi ale lui X, astfel încât fiecare element din X aparţine, cel puţin, unei submulţimi din $\mathcal{F}$:

$$X = \cup_{S \in \mathcal{F}} S.$$

Spunem că o submulţime $S \in \mathcal{F}$ ***acoperă*** elementele sale. Problema este de a găsi o submulţime de dimensiune minimă $\mathcal{C} \subseteq \mathcal{F}$ ai cărei membri acoperă toată mulţimea X:

$$X = \cup_{S \in \mathcal{C}} S. \tag{37.8}$$

Spunem că orice $\mathcal{C}$ care satisface ecuaţia (37.8) ***acoperă*** pe X. Figura 37.3 ilustrează problema.

Problema acoperirii mulţimii este o abstractizare a multor probleme combinatoriale uzuale. De exemplu, să presupunem că X reprezintă o mulţime de deprinderi necesare pentru a rezolva o problemă şi că avem la dispoziţie o mulţime de oameni pentru a lucra asupra problemei. Dorim să formăm un comitet care să conţină cât mai puţini oameni, astfel încât, pentru fiecare deprindere solicitată din X, există un membru al comitetului care să aibă acea deprindere. În versiunea decizională a problemei acoperirii mulţimii întrebăm dacă există sau nu o acoperire de dimensiune cel mult k, unde k este un parametru adiţional specificat în instanţa problemei. Versiunea decizională a problemei este NP-completă, aşa cum se cere să demonstraţi în exerciţiul 37.3-2.

Un algoritm de aproximare greedy

Metoda greedy selectează, la fiecare pas, mulţimea S care acoperă cele mai multe elemente rămase încă neacoperite.

ACOPERIRE-MULŢIME-GREEDY$(X, \mathcal{F})$
1: $U \leftarrow X$
2: $C \leftarrow \emptyset$
3: **cât timp** $U \neq \emptyset$ **execută**
4: selectează un $S \in \mathcal{F}$ care maximizează $|S \cap U|$
5: $U \leftarrow U - S$
6: $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
7: **returnează** $\mathcal{C}$

În exemplul din figura 37.3, algoritmul ACOPERIRE-MULŢIME-GREEDY adaugă la mulţimea $\mathcal{C}$ mulţimile S_1, S_4, S_5 şi S_3, în această ordine.

Algoritmul funcţionează după cum urmează. Mulţimea U conţine, la fiecare pas, mulţimea elementelor rămase neacoperite. Mulţimea $\mathcal{C}$ conţine acoperirea în curs de construire. Linia 4 este pasul în care se ia decizia greedy. Este aleasă o submulţime S care acoperă un număr cât mai mare posibil de elemente neacoperite (cu legăturile rupte în mod arbitrar). După ce S este selectat, elementele sale sunt şterse din U şi S este adăugat la $\mathcal{C}$. Când algoritmul se termină, mulţimea $\mathcal{C}$ conţine o subfamilie a lui $\mathcal{F}$ care îl acoperă pe X.

Algoritmul ACOPERIRE-MULŢIME-GREEDY poate fi implementat cu uşurinţă pentru a se executa în timp polinomial în $|X|$ şi $|\mathcal{F}|$. Deoarece numărul de iteraţii ale buclei din liniile 3–6 este cel mult $\min(|X|, |\mathcal{F}|)$ şi corpul buclei poate fi implementat pentru a se executa în timpul $O(|X||\mathcal{F}|)$, există o implementare care se execută în timpul $O(|X||\mathcal{F}| \min(|X|, |\mathcal{F}|))$. Exerciţiul 37.3-3 cere un algoritm în timp liniar.

Analiză

Vom arăta că algoritmul greedy returnează o acoperire care nu este mult prea mare faţă de o acoperire optimală. Pentru convenienţă, în acest capitol vom nota cu $H(d)$ al d-lea număr armonic $H_d = \sum_{i=1}^{d} 1/i$ (vezi secţiunea 3.1).

Teorema 37.4 Algoritmul ACOPERIRE-MULŢIME-GREEDY are o margine a raportului egală cu

$$H(\max\{|S| : S \in \mathcal{F}\}).$$

Demonstraţie. Demonstraţia urmăreşte atribuirea unui cost fiecărei mulţimi selectate de algoritm, distribuirea acestui cost asupra elementelor acoperite pentru prima dată şi apoi utilizarea acestor costuri pentru a deduce relaţia dorită între dimensiunea unei acoperiri optimale $\mathcal{C}^\star$ şi dimensiunea acoperirii $\mathcal{C}$ returnată de algoritm. Fie S_i a i-a mulţime selectată de algoritmul ACOPERIRE-MULŢIME-GREEDY; algoritmul induce un cost egal cu 1 când adaugă pe S_i la $\mathcal{C}$. Împrăştiem acest cost al selectării lui S_i în mod echilibrat între elementele acoperite pentru prima dată de S_i. Fie c_x costul alocat elementului x, pentru fiecare $x \in X$. Fiecare element primeşte un cost o singură dată, când este acoperit pentru prima dată. Dacă x este acoperit pentru prima dată de S_i, atunci

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

Algoritmul găseşte o soluţie $\mathcal{C}$ de cost total $|\mathcal{C}|$ şi acest cost a fost împrăştiat între elementele lui X. Prin urmare, deoarece acoperirea optimală $\mathcal{C}^\star$ acoperă, de asemenea, pe X, avem

$$|\mathcal{C}| = \sum_{x \in X} c_x \leq \sum_{S \in \mathcal{C}^\star} \sum_{x \in S} c_x. \tag{37.9}$$

Restul demonstraţiei se bazează pe următoarea inegalitate, pe care o vom demonstra pe scurt. Pentru orice mulţime S aparţinând familiei $\mathcal{F}$,

$$\sum_{x \in S} c_x \leq H(|S|). \tag{37.10}$$

Din inegalităţile (37.9) şi (37.10) urmează că

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^\star} H(|S|) \leq |\mathcal{C}^\star| \cdot H(\max\{|S| : S \in \mathcal{F}\}),$$

ceea ce demonstrează teorema. Astfel, rămâne de demonstrat inegalitatea (37.10). Pentru orice mulţime $S \in \mathcal{F}$ şi $i = 1, 2, \ldots, |\mathcal{C}|$, fie

$$u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$$

numărul de elemente ale lui S care rămân neacoperite după ce S_1, S_2, $\ldots$, S_i au fost selectate de algoritm. Definim $u_0 = |S|$ ca reprezentând numărul de elemente din S iniţial neacoperite. Fie k cel mai mic indice astfel încât $u_k = 0$. Prin urmare, fiecare element din S este acoperit de către cel puţin una din mulţimile S_1, S_2, $\ldots$, S_k. Atunci, $u_{i-1} \geq u_i$ şi $u_{i-1} - u_i$ elemente din S sunt acoperite pentru prima dată de S_i, pentru $i = 1, 2, \ldots, k$. Astfel,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

Observăm că

$$|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1},$$

deoarece selecţia greedy S_i garantează că S nu poate acoperi mai multe elemente noi decât S_i (în caz contrar, S ar fi fost selectat în locul lui S_i). Prin urmare obţinem

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

Pentru întregii a şi b, unde $a < b$, avem

$$H(b) - H(a) = \sum_{i=a+1}^{b} 1/i \geq (b-a)\frac{1}{b}.$$

Utilizând această inegalitate, obţinem suma telescopică

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) = H(u_0) - H(0) = H(u_0) = H(|S|), \end{aligned}$$

deoarece $H(0) = 0$. Aceasta încheie demonstraţia inegalităţii (37.10). ∎

Corolarul 37.5 ACOPERIRE-MULŢIME-GREEDY are o margine a raportului de $(\ln|X| + 1)$.

Demonstraţie. Se foloseşte inegalitatea (3.12) şi teorema 37.4. ∎

În unele aplicaţii, $\max\{|S| : S \in \mathcal{F}\}$ este o constantă mică, şi, astfel, soluţia returnată de algoritmul ACOPERIRE-MULŢIME-GREEDY este mai mare cel mult de o constantă de ori decât acoperirea optimală. O astfel de aplicaţie apare în situaţia în care această euristică este folosită pentru obţinerea unei acoperiri cu vârfuri aproximative pentru un graf ale cărui vârfuri au un grad de cel mult 3. În acest caz, soluţia găsită de ACOPERIRE-MULŢIME-GREEDY este mai mare, cel mult, de $H(3) = 11/6$ ori decât soluţia optimă, o asigurare de performanţă care este uşor superioară celei a algoritmului ACOPERIRE-CU-VÂRFURI-APROX.

Exerciţii

37.3-1 Să considerăm fiecare din următoarele cuvinte ca mulţime de litere: {`arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread`}. Arătaţi ce acoperire produce ACOPERIRE-MULŢIME-GREEDY când sunt rupte legăturile în favoarea cuvântului care apare primul în dicţionar.

37.3-2 Arătaţi, prin reducere, folosind problema acoperirii cu vârfuri, că versiunea decizională a problemei de acoperire a mulţimii este NP-completă.

37.3-3 Arătaţi cum se implementează ACOPERIRE-MULŢIME-GREEDY, astfel încât să se execute în timpul $O(\sum_{S\in\mathcal{F}}|S|)$.

37.3-4 Arătaţi că următoarea formă mai slabă a teoremei 37.4 este trivial adevărată:

$$|\mathcal{C}| \leq |\mathcal{C}^\star| \max\{|S| : S \in \mathcal{F}\}.$$

37.3-5 Creaţi o familie de instanţe ale problemei acoperirii mulţimii care să demonstreze că algoritmul ACOPERIRE-MULŢIME-GREEDY poate returna un număr de soluţii diferite care este funcţie exponenţială de dimensiunea instanţei. (Din faptul că legăturile sunt rupte în mod diferit în selecţia lui S, din linia 4, rezultă soluţii diferite.)

37.4. Problema sumei submulţimii

O instanţă a problemei sumei submulţimii este o pereche (S, t) unde S este o mulţime $\{x_1, x_2, \ldots, x_n\}$ de întregi pozitivi şi t este un întreg pozitiv. Se pune problema dacă există o submulţime a lui S pentru care suma elementelor să fie exact valoarea t. Această problemă este NP-completă (vezi secţiunea 36.5.3).

În aplicaţii practice, apare problema de optimizare asociată cu această problemă de decizie. În problema de optimizare, dorim să găsim o submulţime a mulţimii $\{x_1, x_2, \ldots, x_n\}$ a cărei sumă este cât mai mare posibil, dar nu mai mare decât t. De exemplu, am putea avea un camion care nu poate transporta mai mult de t kilograme şi n cutii, cutia i cântărind x_i kilograme. Dorim să umplem camionul cât mai mult, dar fără să depăşim greutatea limită dată.

În această secţiune, prezentăm un algoritm în timp exponenţial pentru această problemă de optimizare şi, apoi, vom arăta cum se poate modifica algoritmul, astfel încât să devină o

schemă de aproximare în timp polinomial. (Să ne reamintim că o schemă de aproximare în timp polinomial are un timp de execuţie polinomial în $1/\epsilon$ şi n.)

Un algoritm în timp exponenţial

Dacă L este o listă de întregi pozitivi şi x este un alt întreg pozitiv, vom nota prin $L+x$ lista de întregi construită pe baza listei L prin însumarea fiecărui element din L cu valoarea x. De exemplu, dacă $L = \langle 1, 2, 3, 5, 9\rangle$, atunci $L + 2 = \langle 3, 4, 5, 7, 11\rangle$. De asemenea, vom folosi această notaţie şi pentru mulţimi:

$$S + x = \{s + x : s \in S\}.$$

Folosim o procedură auxiliară INTERCLASEAZĂ-LISTE(L, L') care returnează lista sortată obţinută prin interclasarea listelor sortate date L şi L'. La fel ca procedura INTERCLASEAZĂ folosită în secţiunea 1.3.1, INTERCLASEAZĂ-LISTE se execută în timp $O(|L| + |L'|)$. (Nu vom prezenta aici pseudocodul pentru INTERCLASEAZĂ-LISTE.) Procedura SUMA-SUBMULŢIMII-EXACTĂ are ca date de intrare o mulţime $S = \{x_1, x_2, \ldots, x_n\}$ şi o valoare ţintă t.

SUMA-SUBMULŢIMII-EXACTĂ(S, t)
1: $n \leftarrow |S|$
2: $L_0 \leftarrow \langle 0\rangle$
3: **pentru** $i \leftarrow 1, n$ **execută**
4: $\quad L_i \leftarrow$ INTERCLASEAZĂ-LISTE$(L_{i-1}, L_{i-1} + x_i)$
5: $\quad$ şterge din L_i toate elementele mai mari decât t
6: **returnează** cel mai mare element din L_n

Fie P_i, mulţimea tuturor valorilor care se pot obţine prin selectarea unei submulţimi (posibil vide) a mulţimii $\{x_1, x_2, \ldots, x_i\}$ şi însumarea membrilor săi. De exemplu, dacă $S = \{1, 4, 5\}$, atunci

$$\begin{aligned} P_1 &= \{0, 1\}, \\ P_2 &= \{0, 1, 4, 5\}, \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\}. \end{aligned}$$

Fiind dată identitatea

$$P_i = P_{i-1} \cup (P_{i-1} + x_i), \tag{37.11}$$

putem demonstra, prin inducţie după i (vezi exerciţiul 37.4-1), că lista L_i este o listă sortată care conţine toate elementele lui P_i a căror valoare nu este mai mare decât t. Deoarece lungimea lui L_i poate fi cel mult 2^i, SUMA-SUBMULŢIMII-EXACTĂ este, în general, un algoritm în timp exponenţial, deşi este un algoritm în timp polinomial în cazurile speciale în care t este polinom în $|S|$ sau toate numerele din S sunt mărginite de un polinom în $|S|$.

O schemă de aproximare în timp complet polinomial

Pentru problema sumei submulţimii putem deduce o schemă de aproximare în timp complet polinomial dacă "filtrăm" fiecare listă L_i după ce este creată. Folosim un parametru de filtrare δ, astfel încât $0 < \delta < 1$. A ***filtra*** o listă L prin δ înseamnă a şterge din L cât mai multe elemente

este posibil, astfel încât, dacă L' este rezultatul filtrării lui L, atunci, pentru fiecare element y care a fost şters din L, există un element $z \leq y$ în L', astfel încât

$$\frac{y - z}{y} \leq \delta$$

sau, echivalent,

$$(1 - \delta)y \leq z \leq y.$$

Astfel, ne putem gândi la z ca "reprezentând" y în noua listă L'. Fiecare y este reprezentat de un z, astfel încât eroarea relativă a lui z, relativ la y, este cel mult δ. De exemplu, dacă δ =0,1 şi

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle,$$

atunci putem filtra L pentru a obţine

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle,$$

unde valoarea ştearsă 11 este reprezentată de 10, valorile şterse 21 şi 22 sunt reprezentate de 20, şi valoarea ştearsă 24 este reprezentată de 23. Este important să reţinem că fiecare element al versiunii filtrate a listei este, de asemenea, un element al versiunii iniţiale a listei. Filtrarea unei liste poate micşora, în mod dramatic, numărul elementelor din listă, păstrând, în acelaşi timp, o valoare reprezentativă apropiată (şi ceva mai mică) pentru fiecare element şters din listă.

Următoarea procedură filtrează o listă de intrare $L = \langle y_1, y_2, \ldots, y_m \rangle$ în timpul $\Theta(m)$, presupunând că L este sortată în ordine nedescrescătoare. Rezultatul returnat de procedură este o listă filtrată, sortată.

```
FILTREAZĂ(L, δ)
1: m ← |L|
2: L' ← ⟨y_1⟩
3: ultimul ← y_1
4: pentru i ← 2, m execută
5:    dacă ultimul < (1 − δ)y_i atunci
6:       adaugă y_i la sfârşitul lui L'
7:       ultimul ← y_i
8: returnează L'
```

Elementele lui L sunt parcurse în ordine crescătoare, iar un număr este inserat în lista returnată L' doar dacă este primul element al lui L sau dacă nu poate fi reprezentat de cel mai recent număr inserat în L'.

Fiind dată procedura FILTREAZĂ, putem construi schema noastră de aproximare după cum urmează. Această procedură primeşte, la intrare, o mulţime $S = \{x_1, x_2, \ldots, x_n\}$ formată din n întregi (în ordine arbitrară), un întreg ţintă t şi un "parametru de aproximare" ϵ, unde $0 < \epsilon < 1$.

SUMA-SUBMULŢIMII-APROX(S, t, ϵ)
1: $n \leftarrow |S|$
2: $L_0 \leftarrow \langle 0 \rangle$
3: **pentru** $i \leftarrow 1, n$ **execută**
4: $L_i \leftarrow$ INTERCLASEAZĂ-LISTE$(L_{i-1}, L_{i-1} + x_i)$
5: $L_i \leftarrow$ FILTREAZĂ$(L_i, \epsilon/n)$
6: şterge din L_i toate elementele mai mari decât t
7: fie z cel mai mare element din L_n
8: **returnează** z

Linia 2 iniţializează lista L_0 la lista având elementul 0. Bucla din liniile 3–6 are efectul calculării lui L_i ca listă sortată, care conţine o versiune filtrată corespunzător a mulţimii P_i, din care toate elementele mai mari decât t au fost şterse. Deoarece L_i este creat din L_{i-1}, trebuie să ne asigurăm că filtrarea repetată nu produce prea mare inacurateţe. Vom vedea că SUMA-SUBMULŢIMII-APROX returnează o aproximaţie corectă, dacă aceasta există.

De exemplu, să presupunem că avem instanţa

$$L = \langle 104, 102, 201, 101 \rangle$$

cu $t = 308$ şi $\epsilon = 0,20$. Parametrul de filtrare δ este $\epsilon/4 = 0,05$. SUMA-SUBMULŢIMII-APROX calculează următoarele valori în liniile indicate:

linia 2: $L_0 = \langle 0 \rangle$,

linia 4: $L_1 = \langle 0, 104 \rangle$,
linia 5: $L_1 = \langle 0, 104 \rangle$,
linia 6: $L_1 = \langle 0, 104 \rangle$,

linia 4: $L_2 = \langle 0, 102, 104, 206 \rangle$,
linia 5: $L_2 = \langle 0, 102, 206 \rangle$,
linia 6: $L_2 = \langle 0, 102, 206 \rangle$,

linia 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,
linia 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,
linia 6: $L_3 = \langle 0, 102, 201, 303 \rangle$,

linia 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
linia 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,
linia 6: $L_4 = \langle 0, 101, 201, 302 \rangle$.

Algoritmul returnează răspunsul $z = 302$, care se încadrează în eroarea de $\epsilon = 20\%$ din răspunsul optimal, $307 = 104 + 102 + 101$; de fapt se încadrează în eroarea 2%.

Teorema 37.6 SUMA-SUBMULŢIMII-APROX este o schemă de aproximare în timp complet polinomial pentru problema sumei submulţimii.

Demonstraţie. Operaţiile de filtrare a lui L_i din linia 5 şi de ştergere din L_i a tuturor elementelor mai mari decât t conservă proprietatea că fiecare element al lui L_i este, de asemenea,

membru al lui P_i. Prin urmare, valoarea z returnată în linia 8 este, într-adevăr, suma unei anumite submulţimi a lui S. Rămâne să arătăm că nu este mai mică de $(1-\epsilon)$ ori decât o soluţie optimală. (Notaţi că, deoarece problema sumei submulţimii este o problemă de maximizare, ecuaţia (37.2) este echivalentă cu $C^\star(1-\epsilon) \le C$.) Trebuie, de asemenea, să arătăm că algoritmul se execută în timp polinomial.

Pentru a arăta că eroarea relativă a soluţiei returnate este mică, să notăm că, în momentul în care lista L_i este filtrată, introducem o eroare relativă de cel mult ϵ/n între valorile reprezentative care rămân în L_i şi valorile de dinainte de filtrare. Prin inducţie după i, se poate arăta că, pentru fiecare element y din P_i nu mai mare decât t, există un $z \in L_i$, astfel încât

$$(1-\epsilon/n)^i y \le z \le y. \tag{37.12}$$

Dacă $y^\star \in P_n$ reprezintă o soluţie optimală a problemei sumei submulţimii, atunci există un $z \in L_n$, astfel încât

$$(1-\epsilon/n)^n y^\star \le z \le y^\star; \tag{37.13}$$

cea mai mare valoare z de acest fel este valoarea returnată de SUMA-SUBMULŢIMII-APROX. Deoarece se poate arăta că

$$\frac{d}{dn}\left(1-\frac{\epsilon}{n}\right)^n > 0,$$

funcţia $(1-\epsilon/n)^n$ creşte în funcţie de n, şi, astfel, $n > 1$ implică

$$1-\epsilon < (1-\epsilon/n)^n,$$

şi de aici,

$$(1-\epsilon)y^\star \le z.$$

Prin urmare, valoarea z returnată de SUMA-SUBMULŢIMII-APROX nu este mai mică de $1-\epsilon$ ori decât soluţia optimală $y^\star$.

Pentru a arăta că aceasta este o schemă de aproximare în timp complet polinomial, deducem o margine a lungimii lui L_i. După filtrare, două elemente succesive z şi z' din L_i trebuie să verifice relaţia $z/z' > 1/(1-\epsilon/n)$. Adică, ele trebuie să difere printr-un factor cel puţin egal cu $1/(1-\epsilon/n)$. Prin urmare, numărul elementelor din fiecare L_i este cel mult

$$\log_{1/(1-\epsilon/n)} t = \frac{\ln t}{-\ln(1-\epsilon/n)} \le \frac{n \ln t}{\epsilon},$$

pe baza ecuaţiei (2.10). Această margine este polinomială în numărul n de valori de intrare, în numărul de biţi $\lg t$ necesari pentru reprezentarea lui t şi în $1/\epsilon$. Deoarece timpul de execuţie al procedurii SUMA-SUBMULŢIMII-APROX este polinomial în lungimea lui L_i, SUMA-SUBMULŢIMII-APROX este o schemă de aproximare în timp complet polinomial. ∎

Exerciţii

37.4-1 Demonstraţi ecuaţia (37.11).

37.4-2 Demonstraţi ecuaţiile (37.12) şi (37.13).

37.4-3 Cum aţi modifica schema de aproximare prezentată în această secţiune pentru a găsi o aproximare bună pentru cea mai mică valoare, nu mai mică decât t, şi care este suma unei submulţimi a listei de intrare?

Probleme

37-1 Împachetarea
Să presupunem că se dă o mulţime de n obiecte, unde dimensiunea s_i a celui de al i-lea obiect satisface $0 < s_i < 1$. Dorim să împachetăm toate obiectele într-un număr minim de pachete de dimensiune 1. Fiecare pachet poate conţine orice submulţime de obiecte având dimensiunea totală cel mult egală cu 1.

a. Demonstraţi că problema determinării numărului minim de pachete necesare este NP-dificilă. (*Indicaţie:* Reduceţi problema la problema sumei submulţimii.)

Euristica ***primei potriviri*** ia fiecare obiect, pe rând, şi îl plasează în primul pachet în care încape. Fie $S = \sum_{i=1}^{n} s_i$.

b. Argumentaţi că numărul optim de pachete necesare este cel puţin $\lceil S \rceil$.

c. Argumentaţi că euristica primei potriviri lasă, cel mult, un pachet plin mai puţin decât pe jumătate.

d. Demonstraţi că numărul pachetelor folosite de euristica primei potriviri nu este niciodată mai mare decât $\lceil 2S \rceil$.

e. Demonstraţi că euristica primei potriviri are marginea raportului egală cu 2.

f. Daţi o implementare eficientă a euristicii primei potriviri şi analizaţi timpul de execuţie al acesteia.

37-2 Aproximarea dimensiunii unei clici maxime
Fie $G = (V, E)$ un graf neorientat. Pentru orice $k \geq 1$, definim $G^{(k)}$ graful neorientat $(V^{(k)}, E^{(k)})$, unde $V^{(k)}$ este mulţimea tuturor k-tuplelor ordonate de vârfuri din V, şi $E^{(k)}$ este definit astfel încât $(v_1, v_2, \ldots, v_k)$ este adiacent cu $(w_1, w_2, \ldots, w_k)$ dacă şi numai dacă, pentru fiecare i, $1 \leq i \leq k$, fie vârful v_i este adiacent cu vârful w_i în G, fie $v_i = w_i$.

a. Demonstraţi că dimensiunea clicii maxime din $G^{(k)}$ este egală cu puterea k a dimensiunii clicii maxime din G.

b. Argumentaţi că, dacă există un algoritm de aproximare cu margine constantă a raportului pentru găsirea clicii maxime, atunci există o schemă de aproximare în timp complet polinomial pentru aceeaşi problemă.

37-3 Problema acoperirii ponderate a mulţimii
Să presupunem că generalizăm problema acoperirii mulţimii, astfel încât, fiecărei mulţimi S_i din familia $\mathcal{F}$ i se asociază o pondere w_i şi că ponderea unei acoperiri $\mathcal{C}$ este $\sum_{S_i \in \mathcal{C}} w_i$. Dorim să determinăm o acoperire de pondere minimă. (Secţiunea 37.3 analizează cazul în care $w_i = 1$ pentru toţi i.)

Arătaţi că euristica greedy de rezolvare a problemei acoperirii mulţimii poate fi generalizată în mod natural pentru a produce o soluţie aproximativă pentru orice instanţă a problemei acoperirii ponderate a mulţimii. Arătaţi că euristica astfel produsă are o margine a raportului de $H(d)$, unde d este dimensiunea maximă a oricărei mulţimi S_i.

Note bibliografice

În domeniul algoritmilor de aproximare există o literatură bogată. Aţi putea începe cu studiul lucrării lui Garey şi Johnson, [79]. Papadimitriou şi Steiglitz [154] au, de asemenea, o prezentare excelentă a algoritmilor de aproximare. Lawler, Lenstra, Rinnooy Kan şi Shmoys [133] prezintă o tratare extensivă a problemei comis-voiajorului.

Papadimitriou şi Steiglitz atribuie algoritmul ACOPERIRE-CU-VÂRFURI-APROX lui F. Gavril şi M. Yannakakis. Algoritmul PCV-TUR-APROX apare într-un articol excelent al lui Rosenkrantz, Stearns şi Lewis [170]. Teorema 37.3 se datorează lui Sahni şi Gonzales [172]. Analiza euristicii greedy pentru problema acoperirii mulţimii este modelată după demonstraţia unui rezultat mai general, publicată de Chvátal [42]; acest rezultat de bază, aşa cum este prezentat aici, este datorat lui Johnson [113] şi Lovász [141]. Algoritmul SUMA-SUBMULŢIMII-APROX şi analiza acestuia sunt uşor modelate după algoritmii de aproximare similari pentru problema rucsacului şi problema sumei submulţimilor, de Ibarra şi Kim [111].

Bibliografie

[1] Milton Abramowitz şi Irene A. Stegun, editori. *Handbook of Mathematical Functions*. Dover, 1965.

[2] G. M. Adel'son-Vel'skii şi E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[3] Leonard M. Adleman, Carl Pomerance, şi Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117:173–206, 1983.

[4] Alfred V. Aho, John E. Hopcroft, şi Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[5] Alfred V. Aho, John E. Hopcroft, şi Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[6] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, şi Robert E. Tarjan. Faster algorithms for the shortest path problem. Technical Report 193, MIT Operations Research Center, 1988.

[7] Howard H. Aiken şi Grace M. Hopper. The automatic sequence controlled calculator. În Brian Randell, editor, *The Origins of Digital Computers*, p. 203–222. Springer-Verlag, third edition, 1982.

[8] M. Ajtai, J. Komlós, şi E. Szemerédi. An $O(n \log n)$ sorting network. În *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, p. 1–9, 1983.

[9] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

[10] Richard J. Anderson şi Gary L. Miller. Deterministic parallel list ranking. În John H. Reif, editor, *1988 Aegean Workshop on Computing*, volumul 319 din *Lecture Notes in Computer Science*, p. 81–90. Springer-Verlag, 1988.

[11] Richard J. Anderson şi Gary L. Miller. A simple randomized parallel algorithm for list-ranking. Unpublished manuscript, 1988.

[12] Tom M. Apostol. *Calculus*, volumul 1. Blaisdell Publishing Company, second edition, 1967.

[13] A. J. Atrubin. A one-dimensional real-time iterative multiplier. *IEEE Transactions on Electronic Computers*, EC-14(1): 394–399, 1965.

[14] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, second edition, 1988.

[15] Eric Bach. Private communication, 1989.

[16] Eric Bach. Number-theoretic algorithms. În *Annual Review of Computer Science*, volumul 4, p. 119–172. Annual Reviews, Inc., 1990.

[17] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[18] R. Bayer şi E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[19] Paul W. Beame, Stephen A. Cook, şi H. James Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15(4):994–1003, 1986.

[20] Pierre Beauchemin, Gilles Brassard, Claude Crépeau, Claude Goutier, şi Carl Pomerance. The generation of random numbers that are probably prime. *Journal of Cryptology*, 1:53–64, 1988.

[21] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[22] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[23] Michael Ben-Or. Lower bounds for algebraic computation trees. În *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, p. 80–86, 1983.

[24] Jon L. Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.

[25] Jon L. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[26] Jon L. Bentley, Dorothea Haken, şi James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

[27] William H. Beyer, editor. *CRC Standard Mathematical Tables*. The Chemical Rubber Company, 1984.

[28] Patrick Billingsley. *Probability and Measure*. John Wiley & Sons, second edition, 1986.

[29] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, şi Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[30] Béla Bollobás. *Random Graphs*. Academic Press, 1985.

[31] J. A. Bondy şi U. S. R. Murty. *Graph Theory with Applications*. American Elsevier, 1976.

[32] Robert S. Boyer şi J. Strother Moore. A fast string-searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[33] Gilles Brassard şi Paul Bratley. *Algorithmics: Theory and Practice*. Prentice-Hall, 1988.

[34] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21 (2):201–206, 1974.

[35] Richard P. Brent. An improved Monte Carlo factorization algorithm. *BIT*, 20(2):176–184, 1980.

[36] Mark R. Brown. *The Analysis of a Practical and Nearly Optimal Priority Queue*. Teză de doctorat, Computer Science Department, Stanford University, 1977. Technical Report STAN-CS-77–600.

[37] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing*, 7(3):298–319, 1978.

[38] Arthur W. Burks, editor. *Theory of Self Reproducing Automata*. University of Illinois Press, 1966.

[39] Joseph J. F. Cavanagh. *Digital Computer Arithmetic*. McGraw-Hill, 1984.

[40] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–507, 1952.

[41] Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes*. Springer-Verlag, 1974.

[42] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[43] V. Chvátal, D. A. Klarner, şi D. E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72–292, Computer Science Department, Stanford University, 1972.

[44] Alan Cobham. The intrinsic computational difficulty of functions. În *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science*, p. 24–30. North-Holland, 1964.

[45] H. Cohen şi H. W. Lenstra, Jr. Primality testing and jacobi sums. *Mathematics of Computation*, 42(165):297–330, 1984.

[46] Richard Cole. Parallel merge sort. În *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, p. 511–516. IEEE Computer Society, 1986.

[47] Richard Cole şi Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[48] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[49] Stephen Cook. The complexity of theorem proving procedures. În *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, p. 151–158, 1971.

[50] Stephen Cook, Cynthia Dwork, şi Rüdiger Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[51] James W. Cooley şi John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90): 297–301, April 1965.

[52] Don Coppersmith şi Shmuel Winograd. Matrix multiplication via arithmetic progressions. În *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, p. 1–6, 1987.

[53] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[54] Whitfield Diffie şi Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[55] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[56] John D. Dixon. Factorization and primality tests. *The American Mathematical Monthly*, 91 (6):333–352, 1984.

[57] Alvin W. Drake. *Fundamentals of Applied Probability Theory*. McGrawHill, 1967.

[58] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, şi Robert E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

[59] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, şi Robert E. Tarjan. Making data structures persistent. În *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, p. 109–121, 1986.

[60] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*, volumul 10 din *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1987.

[61] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[62] Jack Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1:126–136, 1971.

[63] Jack Edmonds şi Richard M. Karp. Theoretical improvements in the algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[64] G. Estrin, B. Gilchrist, şi J. H. Pomerene. A note on high-speed digital multiplication. *IRE Transactions on Electronic Computers*, 5(3):140, 1956.

[65] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[66] William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, third edition, 1968.

[67] M. J. Fischer şi A. R. Meyer. Boolean matrix multiplication and transitive closure. În *Proceedings of the Twelfth Annual Symposium on Switching and Automata Theory*, p. 129–131. IEEE Computer Society, 1971.

[68] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

[69] Robert W. Floyd. Algorithm 245 (TREESORT). *Communications of the ACM*, 7:701, 1964.

[70] Robert W. Floyd şi Ronald L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18(3):165–172, 1975.

[71] Lestor R. Ford, Jr., şi D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[72] Lector R. Ford, Jr., şi Selmer M. Johnson. A tournament problem. *The American Mathematical Monthly*, 66:387–389, 1959.

[73] Steven Fortune şi James Wyllie. Parallelism in random access machines. În *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, p. 114–118, 1978.

[74] Michael L. Fredman şi Michael E. Saks. The cell probe complexity of dynamic data structures. În *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, 1989.

[75] Michael L. Fredman şi Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[76] Harold N. Gabow şi Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[77] Harold N. Gabow şi Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989.

[78] Zvi Galil şi Joel Seiferas. Time-space-optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.

[79] Michael R. Garey şi David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[80] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[81] Alan George şi Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

[82] Andrew V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. Teză de doctorat, Department of Electrical Engineering and Computer Science, MIT, 1987.

[83] Andrew V. Goldberg, Éva Tardos, şi Robert E. Tarjan. Network flow algorithms. Technical Report STAN-CS-89–1252, Computer Science Department, Stanford University, 1989.

[84] Andrew V. Goldberg şi Serge A. Plotkin. Parallel $(\delta + 1)$ coloring of constant-degree graphs. *Information Processing Letters*, 25(4):241–245, 1987.

[85] Andrew V. Goldberg şi Robert E. Tarjan. A new approach to the maximum flow problem. În *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, p. 136–146, 1986.

[86] Shafi Goldwasser şi Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[87] Shafi Goldwasser, Silvio Micali, şi Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[88] Shafi Goldwasser, Silvio Micali, şi Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[89] Gene H. Golub şi Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1983.

[90] G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984.

[91] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132–133, 1972.

[92] R. L. Graham şi Pavol Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[93] Leo J. Guibas şi Robert Sedgewick. A diochromatic framework for balanced trees. În *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, p. 8–21. IEEE Computer Society, 1978.

[94] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.

[95] J. Hartmanis şi R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[96] Frederick J. Hill şi Gerald R. Peterson. *Introduction to Switching Theory and Logical Design*. John Wiley & Sons, second edition, 1974.

[97] C. A. R. Hoare. Algorithm 63 (partition) and algorithm 65 (find). *Communications of the ACM*, 4(7):321–322, 1961.

[98] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[99] W. Hoeffding. On the distribution of the number of successes in independent trials. *Annals of Mathematical Statistics*, 27:713–721, 1956.

[100] Micha Hofri. *Probabilistic Analysis of Algorithms*. Springer-Verlag, 1987.

[101] John E. Hopcroft şi Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[102] John E. Hopcroft şi Robert E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[103] John E. Hopcroft şi Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[104] John E. Hopcroft şi Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[105] Ellis Horowitz şi Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[106] T. C. Hu şi M. T. Shing. *Computation of Matrix Chain Products. Part I. SIAM Journal on Computing*, 11(2): 362–373, 1982. T. C. Hu şi M. T. Shing. *Computation of Matrix Chain Products. Part II. SIAM Journal on Computing*, 13(2): 228–251, 1984.

[107] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[108] Kai Hwang. Computer Arithmetic: *Principles, Architecture, and Design*. John Wiley & Sons, 1979.

[109] Kai Hwang şi Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[110] Kai Hwang şi Doug DeGroot. *Parallel Processing for Supercomputers and Artificial Intelligence*. McGraw-Hill, 1989.

[111] Oscar H. Ibarra şi Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, 1975.

[112] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2:18–21, 1973.

[113] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[114] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[115] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[116] Richard M. Karp. Reducibility among combinatorial problems. În Raymond E. Miller şi James W. Thatcher, editori, *Complexity of Computer Computations*, p. 85–103. Plenum Press, 1972.

[117] Richard M. Karp şi Michael O. Rabin. Efficient randomized patternmatching algorithms. Technical Report TR-31–81, Aiken Computation Laboratory, Harvard University, 1981.

[118] Richard M. Karp şi Vijaya Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division (EECS), University of California, Berkeley, 1988.

[119] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[120] D. G. Kirkpatrick şi R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15(2):287–299, 1986.

[121] Donald E. Knuth. *Fundamental Algorithms*, volumul 1 din *The Art of Computer Programming*. Addison-Wesley, 1968. Second edition, 1973.

[122] Donald E. Knuth. *Seminumerical Algorithms*, volumul 2 din *The Art of Computer Programming*. Addison-Wesley, 1969. Second edition, 1981.

[123] Donald E. Knuth. *Sorting and Searching*, volumul 3 din *The Art of Computer Programming*. Addison-Wesley, 1973.

[124] Donald E. Knuth. Big omicron and big omega and big theta. *ACM SIGACT News*, 8(2):18–23, 1976.

[125] Donald E. Knuth, James H. Morris, Jr., şi Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[126] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 1970.

[127] Bernhard Korte şi László Lovász. Mathematical structures underlying greedy algorithms. În F. Gecseg, editor, *Fundamentals of Computation Theory*, numărul 117 din *Lecture Notes in Computer Science*, p. 205–209. Springer-Verlag, 1981.

[128] Bernhard Korte şi László Lovász. Structural properties of greedoids. *Combinatorica*, 3:359–374, 1983.

[129] Bernhard Korte şi László Lovász. Greedoids – a structural framework for the greedy algorithm. În W. Pulleybank, editor, *Progress in Combinatorial Optimization*, p. 221–243. Academic Press, 1984.

[130] Bernhard Korte şi László Lovász. Greedoids and linear objective functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(2):229–238, 1984.

[131] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[132] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.

[133] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, şi D. B. Shmoys, editori. *The Traveling Salesman Problem*. John Wiley & Sons, 1985.

[134] C. Y. Lee. An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

[135] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Networks and Algorithms*. Morgan-Kaufmann, in preparation.

[136] Debra A. Lelewer şi Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[137] H. W. Lenstra, Jr. Factoring integers with elliptic curves. *Annals of Mathematics*, 126:649–673, 1987.

[138] L. A. Levin. Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. În limba rusă.

[139] Harry R. Lewis şi Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[140] C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968.

[141] László Lovász. *On the ratio of optimal integral and fractional covers*. Discrete Mathematics, 13:383–390, 1975.

[142] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989.

[143] William J. Masek şi Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[144] Kurt Mehlhorn. *Sorting and Searching*, volumul 1 din *Data Structures and Algorithms*. Springer-Verlag, 1984.

[145] Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volumul 2 din *Data Structures and Algorithms*. Springer-Verlag, 1984.

[146] Kurt Mehlhorn. *Multidimensional Searching and Computational Geometry*, volumul 3 din *Data Structures and Algorithms*. Springer-Verlag, 1984.

[147] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[148] Louis Monier. *Algorithmes de Factorisation D'Entiers*. Teză de doctorat, L'Université Paris-Sud, Centre D'Orsay, 1980.

[149] Louis Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, 12(1):97–108, 1980.

[150] Edward F. Moore. The shortest path through a maze. În *Proceedings of the International Symposium on the Theory of Switching*, p. 285–292. Harvard University Press, 1959.

[151] Ivan Niven şi Herbert S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, fourth edition, 1980.

[152] Yu. Ofman. On the algorithmic complexity of discrete functions. *Soviet Physics Doklady*, 7(7):589–591, 1963. Traducere în limba engleză.

[153] Alan V. Oppenheim şi Alan S. Willsky, cu Ian T. Young. *Signals and Systems*. Prentice-Hall, 1983.

[154] Christos H. Papadimitriou şi Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[155] Michael S. Paterson, 1974. Unpublished lecture, Ile de Berder, France.

[156] J. M. Pollard. A Monte Carlo method for factorization. BIT, 15:331–334, 1975.

[157] Carl Pomerance. On the distribution of pseudoprimes. *Mathematics of Computation*, 37(156):587–593, 1981.

[158] Carl Pomerance. The quadratic sieve factoring algorithm. În T. Beth, N. Cot, şi I. Ingemarrson, editori, *Advances in Cryptology*, volumul 209 din *Lecture Notes in Computer Science*, p. 169–182. Springer-Verlag, 1984.

[159] Carl Pomerance, editor. *Proceedings of the AMS Symposia in Applied Mathematics: Computational Number Theory and Cryptography*. American Mathematical Society, 1992.

[160] Franco P. Preparata şi Micheal Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[161] William H. Press, Brian P. Flannery, Saul A. Teukolsky, şi William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 1986.

[162] William H. Press, Brian P. Flannery, Saul A. Teukolsky, şi William T. Vetterling. Numerical Recipes in C. Cambridge University Press, 1988.

[163] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[164] Paul W. Purdom, Jr., şi Cynthia A. Brown. *The Analysis of Algorithms*. Holt, Rinehart, şi Winston, 1985.

[165] Michael O. Rabin. Probabilistic algorithms. În J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, p. 21–39. Academic Press, 1976.

[166] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

[167] Edward M. Reingold, Jürg Nievergelt, şi Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[168] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. Progress in Mathematics. Birkhäuser, 1985.

[169] Ronald L. Rivest, Adi Shamir, şi Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. Vezi şi U.S. Patent 4,405,829.

[170] D. J. Rosenkrantz, R. E. Stearns, şi P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6:563–581, 1977.

[171] Y. A. Rozanov. *Probability Theory: A Concise Course*. Dover, 1969.

[172] S. Sahni şi T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23:555–565, 1976.

[173] John E. Savage. *The Complexity of Computing*. John Wiley & Sons, 1976.

[174] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, 1978.

[175] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[176] Michael I. Shamos şi Dan Hoey. Geometric intersection problems. În *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, p. 208–215. IEEE Computer Society, 1976.

[177] Daniel D. Sleator şi Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[178] Daniel D. Sleator şi Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[179] Joel Spencer. *Ten Lectures on the Probabilistic Method*. Regional Conference Series on Applied Mathematics (No. 52). SIAM, 1987.

[180] Staff of the Computation Laboratory. *Description of a Relay Calculator*, volumul XXIV din *The Annals of the Computation Laboratory of Harvard University*. Harvard University Press, 1949.

[181] Gilbert Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, 1986.

[182] Gilbert Strang. *Linear Algebra and Its Applications*. Harcourt Brace Jovanovich, third edition, 1988.

[183] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[184] T. G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.

[185] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[186] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[187] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2): 110–127, 1979.

[188] Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[189] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[190] Robert E. Tarjan şi Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[191] Robert E. Tarjan şi Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[192] George B. Thomas, Jr., şi Ross L. Finney. *Calculus and Analytic Geometry*. Addison-Wesley, seventh edition, 1988.

[193] Leslie G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4(3):348–355, 1975.

[194] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. În *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, p. 75–84. IEEE Computer Society, 1975.

[195] Uzi Vishkin. Implementation of simultaneous memory address access in models that forbid it. *Journal of Algorithms*, 4(1):45–50, 1983.

[196] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, 1978.

[197] C. S. Wallace. A suggestion for a fast multiplier. *IEEE Transactions on Electronic Computers*, EC-13(1):14–17, 1964.

[198] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[199] A. Weinberger şi J. L. Smith. A one-microsecond adder using onemegacycle circuitry. *IRE Transactions on Electronic Computers*, EC-5(2), 1956.

[200] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, 1935.

[201] Herbert S. Wilf. *Algorithms and Complexity*. Prentice-Hall, 1986.

[202] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

[203] S. Winograd. On the algebraic complexity of functions. În *Actes du Congres International des Mathématiciens*, volumul 3, p. 283–288, 1970.

[204] James C. Wyllie. *The Complexity of Parallel Computations*. Teză de doctorat, Department of Computer Science, Cornell University, 1979.

[205] Andrew C.-C. Yao. A lower bound to finding convex hulls. *Journal of the ACM*, 28(4):780–787, 1981.

Index