Introduzione agli algoritmi

Sommario.

Prefazione all'edizione italiana XI

Nota alla seconda edizione XI

Prefazione XVII

1 Introduzione 1

1.1 Algoritmi 1

1.2 Analisi di algoritmi 5

1.3 Progetto di algoritmi 8

1.4 Riepilogo 12

PARTE PRIMA - FONDAMENTI DI MATEMATICA 15

Introduzione 17

2 Ordine di grandezza delle funzioni 21

2.1 Notazione asintotica 23

2.2 Notazioni standard e funzioni comuni 28

3 Sommatorie 39

3.1 Formule e proprietà sulle sommatorie 41

3.2 Definizione di limitazioni sulle sommatorie 45

4 **Ricorrenze** ... 51
- 4.1 Il metodo di sostituzione ... 52
- 4.2 Il metodo iterativo ... 55
- 4.3 Il metodo principale ... 58
- * 4.4 Dimostrazione del teorema principale ... 61

5 **Insiemi e affini** ... 73
- 5.1 Insiemi ... 73
- 5.2 Relazioni ... 77
- 5.3 Funzioni ... 79
- 5.4 Grafi ... 81
- 5.5 Alberi ... 85

6 **Calcolo combinatorio e delle probabilità** ... 93
- 6.1 Calcolo combinatorio ... 93
- 6.2 Calcolo delle probabilità ... 98
- 6.3 Variabili casuali discrete ... 103
- 6.4 Distribuzione geometrica e distribuzione binomiale ... 107
- * 6.5 Code della distribuzione binomiale ... 113
- 6.6 Analisi probabilistica ... 118

PARTE SECONDA - ORDINAMENTO E SELEZIONE ... 128

Introduzione ... 129

7 **Heapsort** ... 133
- 7.1 Heap ... 133
- 7.2 Mantenimento della proprietà dello heap ... 135
- 7.3 Costruzione di uno heap ... 137
- 7.4 L'algoritmo heapsort ... 139
- 7.5 Code con priorità ... 141

8 **Quicksort** ... 145
- 8.1 Descrizione del quicksort ... 145
- 8.2 Prestazioni di quicksort ... 147
- 8.3 Versione randomizzata del quicksort ... 152
- 8.4 Analisi del quicksort ... 154

9 **Ordinamento in tempo lineare** ... 163
- 9.1 Limiti inferiori per l'ordinamento ... 163
- 9.2 Counting sort ... 166
- 9.3 Radix sort ... 168
- 9.4 Bucket sort ... 170

10 **Mediano e selezione** ... 175
- 10.1 Minimo e massimo ... 175
- 10.2 Selezione con tempo medio lineare ... 177
- 10.3 Selezione in tempo lineare nel caso peggiore ... 179

PARTE TERZA - STRUTTURE DI DATI ... 184

Introduzione ... 185

11 **Strutture di dati fondamentali** ... 189
- 11.1 Pile e Code ... 189
- 11.2 Liste concatenate ... 192
- 11.3 Realizzazione di puntatori e oggetti ... 197
- 11.4 Rappresentazione di alberi radicati ... 201

12 **Tabelle hash** ... 207
- 12.1 Tabelle ad indirizzamento diretto ... 207
- 12.2 Tabelle hash ... 209
- 12.3 Funzioni hash ... 213
- 12.4 Indirizzamento aperto ... 219

13 **Alberi binari di ricerca** ... 229
- 13.1 Che cos'è un albero binario di ricerca? ... 229
- 13.2 Interrogazioni su un albero binario di ricerca ... 231
- 13.3 Inserzione e cancellazione ... 235
- * 13.4 Alberi binari di ricerca costruiti in modo casuale ... 238

14 **RB-alberi** ... 247
- 14.1 Proprietà degli RB-alberi ... 247
- 14.2 Rotazioni ... 249
- 14.3 Inserzione ... 251
- 14.4 Cancellazione ... 255

15 Estensione di strutture di dati 263
15.1 Selezione su un insieme dinamico 263
15.2 Come estendere una struttura di dati 268
15.3 Alberi di intervalli 271

PARTE QUARTA - TECNICHE EVOLUTE PER IL PROGETTO E L'ANALISI DI ALGORITMI 278

Introduzione 279

16 Programmazione dinamica 283
16.1 Prodotto di una sequenza di matrici 284
16.2 Elementi di programmazione dinamica 291
16.3 Il problema della più lunga sottosequenza comune 296
16.4 Triangolazione ottima di un poligono 302

17 Algoritmi greedy 311
17.1 Selezione di attività 312
17.2 Strategia greedy: concetti di base 315
17.3 Codici di Huffman 319
* 17.4 Fondamenti teorici dei metodi greedy 326
* 17.5 Un problema di scheduling 332

18 Analisi ammortizzata 337
18.1 Il metodo degli aggregati 338
18.2 Il metodo degli accantonamenti 342
18.3 Il metodo del potenziale 344
18.4 Tabelle dinamiche 348

PARTE QUINTA - STRUTTURE DI DATI EVOLUTE 360

Introduzione 361

19 B-Alberi 363
19.1 Definizione dei B-alberi 366
19.2 Operazioni di base sui B-alberi 369
19.3 Eliminazione di una chiave da un B-albero 376

20 Heap binomiali 383
20.1 Alberi binomiali e heap binomiali 384
20.2 Operazioni su heap binomiali 389

21 Gli heap di Fibonacci 403
21.1 La struttura degli heap di Fibonacci 404
21.2 Operazioni che fondono heap 406
21.3 Decremento di una chiave ed eliminazione di un nodo 413
21.4 Limitazione del grado massimo 417

22 Strutture di dati per insiemi disgiunti 423
22.1 Operazioni su insiemi disgiunti 423
22.2 Rappresentazione a lista concatenata di insiemi disgiunti 426
22.3 Foreste di insiemi disgiunti 429
* 22.4 Analisi dell'unione per rango con compressione dei cammini 432

PARTE SESTA - ALGORITMI SU GRAFI 444

Introduzione 445

23 Algoritmi elementari su grafi 447
23.1 Rappresentazione di grafi 447
23.2 Visita in ampiezza 450
23.3 Visita in profondità 458
23.4 Ordinamento topologico 465
23.5 Componenti fortemente connesse 468

24 Alberi di copertura minimi 477
24.1 Costruzione di un albero di copertura minimo 478
24.2 Gli algoritmi di Kruskal e di Prim 482

25 Cammini minimi con sorgente singola 491
25.1 Cammini minimi e rilassamento 495
25.2 Algoritmo di Dijkstra 503
25.3 Algoritmo di Bellman-Ford 507
25.4 Cammini minimi con sorgente singola in grafi orientati aciclici 511
25.5 Vincoli di differenza e cammini minimi 513

26 Cammini minimi tra tutte le coppie 525
26.1 Cammini minimi e moltiplicazione di matrici 527
26.2 Algoritmo di Floyd-Warshall 532
26.3 Algoritmo di Johnson per grafi sparsi 539
* 26.4 Un contesto generale in cui risolvere problemi di cammini in grafi orientati 543

27 Flusso massimo 551
27.1 Reti di flusso 552
27.2 Il metodo di Ford-Fulkerson 558
27.3 Abbinamento massimo in un grafo bipartito 570
* 27.4 Algoritmi di preflusso 574
* 27.5 Algoritmo lift-to-front 583

PARTE SETTIMA - COMPLEMENTI ED ESTENSIONI 598

Introduzione 599

28 Reti di confrontatori 601
28.1 Reti di confrontatori 601
28.2 Il principio zero-uno 605
28.3 Una rete di ordinamento bitonico 608
28.4 Una rete di fusione 611
28.5 Una rete di ordinamento 613

29 Circuiti aritmetici 619
29.1 Circuiti combinatori 619
29.2 Addizionatori 624
29.3 Circuiti moltiplicatori 634
29.4 Circuiti sequenziali 641

30 Algoritmi per calcolatori paralleli 651
30.1 Salto dei puntatori 654
30.2 Confronto tra algoritmi EREW e algoritmi CRCW 663
30.3 Il Teorema di Brent e l'efficienza rispetto al lavoro 670
* 30.4 Calcolo parallelo dei prefissi efficiente rispetto al lavoro 674
30.5 Risoluzione deterministica dei conflitti 679

31 Operatori sulle matrici 689
31.1 Proprietà delle matrici 689
31.2 Algoritmo di Strassen per la moltiplicazione tra matrici 697
* 31.3 Sistemi algebrici di numeri e moltiplicazione tra matrici booleane 704
31.4 Risoluzione di sistemi di equazioni lineari 708
31.5 Inversione di matrici 720
31.6 Matrici simmetriche definite positive e metodo dei minimi quadrati 724

32 Polinomi e FFT 735
32.1 Rappresentazione di polinomi 737
32.2 DFT e FFT 742
32.3 Realizzazioni efficienti della FFT 749

33 Algoritmi di teoria dei numeri 759
33.1 Nozioni elementari di teoria dei numeri 760
33.2 Massimo comun divisore 765
33.3 Aritmetica modulare 770
33.4 Risoluzione di equazioni lineari modulari 776
33.5 Il teorema cinese del resto 779
33.6 Potenze di un elemento 781
33.7 Crittografia a chiave pubblica RSA 785
* 33.8 Verifica di primalità 791
* 33.9 Scomposizione di interi in fattori primi 797

34 Corrispondenza tra stringhe 805
34.1 Algoritmo ingenuo di corrispondenza tra stringhe 806
34.2 Algoritmo Rabin-Karp 809
34.3 Corrispondenza tra stringhe con gli automi a stati finiti 813
34.4 Algoritmo Knuth-Morris-Pratt 819
* 34.5 Algoritmo Boyer-Moore 825

35 Geometria computazionale 835
35.1 Proprietà dei segmenti 835
35.2 Verifica dell'intersezione di una coppia qualsiasi di segmenti 840
35.3 Calcolo dell'inviluppo convesso 846
35.4 Ricerca della coppia di punti più vicini 854

36 Problemi NP-completi 863
36.1 Tempo polinomiale 864

36.2 Verifica in tempo polinomiale 870
36.3 NP-completezza e riducibilità 875
36.4 Dimostrazioni di NP-completezza 883
36.5 Problemi NP-completi 890

37 Algoritmi approssimati 907
37.1 Il problema della copertura di vertici 909
37.2 Il problema del commesso viaggiatore 911
37.3 Il problema della copertura di un insieme 916
37.4 Il problema della somma di sottoinsieme 920

Bibliografia 927

Indice analitico 937

Prefazione all'edizione italiana

Il termine *algoritmo*, che significa procedimento di calcolo, è una versione moderna del termine latino medievale *algorismus*, che deriva dal nome del matematico usbeco Mohammed ibn-Musa *al-Khowarismi*, vissuto nel IX secolo d.C. e famoso per aver scritto un noto trattato di algebra.

Col termine algoritmo si intende la descrizione precisa di una sequenza di azioni che devono essere eseguite per giungere alla risoluzione di un problema computazionale. La nozione di algoritmo, pur essendo vicina alla nozione di dimostrazione matematica, era già presente nelle origini della civiltà umana ben prima che fosse definita la nozione di dimostrazione. La nozione di *dimostrazione euclidea* fu formalizzata dai matematici greci come una sequenza finita di formule, ottenute per trasformazioni successive a partire da un insieme dato di assiomi, che permettono di provare l'enunciato del teorema. La nozione di algoritmo, invece, si trova già registrata in documenti risalenti al XVII secolo a.C. e conservati nel Museo Britannico. Infatti, i papiri egizi di *Ahmes* contengono una collezione di problemi con le relative soluzioni, ivi compreso un ingegnoso procedimento di moltiplicazione, che Ahmes dichiara di aver in parte copiato da altri papiri anteriori di circa dodici secoli.

Pur se la *teoria* degli algoritmi si è andata assestando nella prima metà del XX secolo, le *tecniche* per *progettare* algoritmi e per *analizzarne* la *correttezza* e l'*efficienza* si sono evolute nella seconda metà del secolo, grazie alla enorme diffusione dei *calcolatori elettronici*.

Ovunque si impieghi un calcolatore elettronico occorrono algoritmi corretti ed efficienti che ne utilizzino al massimo le possibilità. Algoritmi sofisticati e veloci sono usati per controllare il volo di aerei, regolare reazioni nucleari, reperire informazioni in archivi di dati, smistare comunicazioni telefoniche, giocare a scacchi, controllare la produzione di catene di montaggio, tradurre programmi da un certo linguaggio di programmazione ad un altro, progettare nuovi calcolatori, riconoscere immagini, elaborare testi letterari, comporre musica.

Gli algoritmi vengono comunemente descritti tramite *programmi*, che si avvalgono di istruzioni e costrutti dei *linguaggi di programmazione* e che devono essere eseguiti da calcolatori elettronici. Le proprietà degli algoritmi sono però talmente *fondamentali*, *generali* e *robuste* da essere virtualmente *indipendenti* dalle caratteristiche di specifici linguaggi di programmazione o di particolari calcolatori elettronici.

La nozione di algoritmo è inscindibilmente legata a quella di *dato*. Infatti, per risolvere un problema computazionale, occorre organizzare ed elaborare dati. L'algoritmo può essere interpretato come un manipolatore di dati che, a fronte di dati d'ingresso che descrivono il problema da risolvere, produce altri dati di uscita come risultato del problema. Pertanto, è

fondamentale che i dati siano ben organizzati e *strutturati* in modo che l'algoritmo li possa elaborare efficientemente.

Questo libro di testo fornisce sia una moderna introduzione che un esauriente trattato riguardo al progetto e all'analisi di algoritmi, ed è stato scritto da famosi professori e ricercatori del prestigioso Massachusetts Institute of Technology. Il testo presenta molti algoritmi per problemi di indubbia utilità pratica, alcuni dei quali sono trattati molto approfonditamente, pur rendendone accessibili il progetto e l'analisi a lettori, siano essi studenti universitari o professionisti, che abbiano una preparazione matematica a livello di scuola media superiore.

Gli algoritmi sono descritti sia verbalmente che utilizzando un semplice *pseudocodice*, che li rende comprensibili a qualsiasi lettore che abbia una minima esperienza di programmazione svincolandolo dall'uso di uno specifico linguaggio di programmazione. Nel tradurre il testo in italiano, si è posto il dilemma se usare termini scientifici della lingua italiana o inglese. Dove è stato possibile, sono stati usati termini italiani di uso corrente nell'ambito accademico. Nei casi in cui la traduzione avrebbe richiesto l'introduzione di neologismi, sono stati mantenuti i termini inglesi (per esempio *bit*, *array*, *heap*, *hash*) divenuti ormai di uso corrente anche in italiano, perlomeno in ambito accademico. In particolare, le istruzioni dello pseudocodice e i nomi delle variabili e degli algoritmi sono stati lasciati in inglese.

Il testo è suddiviso in sette parti, a loro volta suddivise in capitoli. Ciascun capitolo presenta dapprima le nozioni più semplici e successivamente quelle più difficili, cosí che un lettore inesperto possa limitarsi a leggere le prime. Alcuni paragrafi ed esercizi sono contrassegnati da un asterisco (*). Questi indicano approfondimenti che richiedono la comprensione di tecniche matematiche più complesse.

La Parte I tratta tutte le tecniche matematiche che sono necessarie nel resto del testo. L'unico prerequisito richiesto al lettore è quello di avere una minima dimestichezza con le dimostrazioni che usano l'induzione matematica. Questa parte può essere soltanto sfogliata ad una prima lettura, per ritornarvi successivamente quando risulti necessario capire una certa tecnica matematica usata nel progetto o nell'analisi di un algoritmo trattato nelle Parti II-VII. La Parte II espone i principali algoritmi di ordinamento (*heapsort*, *quicksort*, *counting sort*, *radix sort*, *bucket sort*) e di selezione, mentre la Parte III introduce le principali strutture di dati (*liste*, *code*, *pile*, *tabelle hash*, *alberi di ricerca*, *alberi bilanciati*) atte a realizzare operazioni su insiemi dinamici.

La Parte IV presenta tecniche sofisticate per progettare nuovi algoritmi (*programmazione dinamica*, *greedy*) e per analizzarne l'efficienza (*analisi ammortizzata*). La Parte V tratta strutture di dati evolute (*B-alberi*, *heap binomiali*, *heap di Fibonacci*, *strutture per insiemi disgiunti*), molte delle quali sono state inventate nell'ultimo decennio. La Parte VI espone algoritmi per operare su di una importantissima struttura: i *grafi*. Oltre ad algoritmi di base (*visita in ampiezza o in profondità*, *componenti connesse*), sono considerati algoritmi sia classici che recentissimi per problemi di *ottimizzazione* su grafi (*alberi di copertura*, *cammini minimi*, *flussi massimi*).

La Parte VII contiene complementi ed estensioni del materiale trattato nelle Parti II-VI. Sono introdotti nuovi modelli di calcolo (*reti di confrontatori*, *circuiti combinatori e aritmetici*, *calcolatori paralleli*), sono considerate applicazioni specialistiche (*matrici*, *trasformate di Fourier*, *teoria dei numeri*, *geometria computazionale*, *string matching*), sono trattate alcune limitazioni teoriche di certi problemi, che pare non possano essere risolti in un

lasso di tempo ragionevole (*problemi NP-completi*) e sono considerate tecniche per superare almeno in parte tali limitazioni (*algoritmi approssimati*).

Alan A. Bertossi
Docente di *Metodi per il Trattamento dell'Informazione*
Università degli Studi di Pisa

Pisa, gennaio 1995

Nota alla seconda edizione italiana

In questa seconda edizione il testo è stato minuziosamente rivisto, avvalendosi anche dell'*errata-corrige* messa a disposizione dagli autori su Internet. In un libro di quasi mille pagine è inevitabile che sfuggano degli errori e che vi siano altre imperfezioni dovute alla traduzione, alla preparazione del testo e delle figure, alla lavorazione del libro in generale. Il traduttore non può sempre essere uno specialista del ramo: per questo viene affiancato da un revisore, ma vi è comunque spazio per miglioramenti e correzioni. Gli autori stessi commettono errori o, per motivi magari inesplicabili a loro stessi, non presentano gli argomenti proposti nel modo migliore: in questo libro, il caso più eclatante era forse quello del teorema 12.7, che è stato ora reso più preciso. *Quandoque bonus dormitat Homerus...* anche Omero sembra sonnecchiare talvolta, diceva già Orazio.

Gli studenti (e anche i docenti) spesso si scandalizzano per la quantità di errori, laddove dovrebbero invece sentirsi meno frustrati e in qualche modo rassicurati dalla fallibilità di illustri professori (o colleghi). Ma è soprattutto a beneficio degli studenti che è importante cercare di minimizzare il numero di errori, giacché questi tendono a rendere meno intelligibile il testo e non sempre sono facilmente scoperti dal lettore meno provveduto. Purtroppo, correggere gli errori è un lavoro minuzioso e ingrato, al quale ci siamo sobbarcati con spirito di servizio e cercando di fare del nostro meglio: ci scusiamo fin d'ora se qualche svista è rimasta o se ne sono state introdotte di nuove. Ringraziamo altresì i numerosi studenti che hanno dovuto cimentarsi con la precedente edizione e ci hanno fruttuosamente segnalato errori o oscurità, traendo peraltro da questi spunto per una più profonda comprensione dell'argomento.

È opportuno segnalare che, allo scopo di mitigare la ben nota confusione tra alberi pieni, completi e heap, abbiamo chiamato ***albero pienamente binario*** il ***full binary tree***, precedentemente tradotto con ***albero binario pieno***. Per il resto, la terminologia è rimasta invariata.

Mauro Torelli e *Carlo Mereghetti*
Dipartimento di *Scienze dell'informazione*
Università di Milano

Milano, maggio 1999

Prefazione

Questo libro fornisce un'ampia introduzione allo studio moderno degli algoritmi. Presenta molti algoritmi e li tratta in modo approfondito; inoltre rende la loro progettazione e la loro analisi accessibili a lettori di qualunque livello. Si è cercato di mantenere le spiegazioni a livello elementare senza sacrificare l'ampiezza della trattazione o il rigore matematico.

Ogni capitolo presenta un algoritmo, una tecnica di progettazione, un'area di applicazione o un argomento attinente. Gli algoritmi sono descritti con uno pseudocodice pensato per essere leggibile da chiunque abbia fatto un po' di programmazione; il loro funzionamento è illustrato da numerose figure. Poiché l'*efficienza* va considerata determinante nei criteri di progettazione di un algoritmo, è inclusa un'attenta analisi dei tempi di esecuzione.

Il testo è rivolto soprattutto a studenti di corsi universitari su algoritmi e strutture di dati. Dato che si discutono sia le questioni tecniche che gli aspetti matematici presenti nella progettazione di algoritmi, il libro può essere utilizzato proficuamente anche da tecnici e professionisti.

Al docente

Questo libro è stato pensato per essere versatile e completo ed è quindi utilizzabile in una varietà di corsi, ovunque ci si occupi di algoritmi e strutture di dati. Dato che si fornisce molto più materiale di quello che può comprendere un corso, il docente dovrebbe considerare il libro come testo di riferimento da cui prendere e selezionare il materiale che meglio si adatta al corso che intende svolgere, utilizzando solo i capitoli necessari. Ogni capitolo contiene tutte le nozioni necessarie alla sua comprensione e presenta prima gli argomenti più semplici e poi quelli più complicati. In un corso di diploma si potranno usare solo i primi paragrafi di un capitolo, in un corso di laurea si potrà usare l'intero capitolo.

Vengono presentati oltre 900 esercizi e più di 120 problemi. Ogni paragrafo termina con esercizi e ciascun capitolo termina con problemi. Generalmente gli esercizi sono domande brevi per verificare l'apprendimento del materiale; alcuni sono semplici esercizi di autoverifica, mentre altri possono essere dati come compito per casa. I problemi propongono casi più elaborati e spesso introducono nuovo materiale; generalmente consistono di diverse domande che conducono lo studente per passi fino alla soluzione.

I paragrafi e gli esercizi adatti a studenti più maturi sono segnalati con un asterisco (*). Un paragrafo con asterisco non è necessariamente più difficile di uno senza, ma può richiedere la conoscenza di aspetti matematici più complessi. Analogamente, gli esercizi con asterisco possono richiedere una preparazione di base più completa o una creatività superiore alla media.

Allo studente

Si spera che questo libro fornisca al lettore una piacevole introduzione alle problematiche degli algoritmi che si è cercato di rendere interessanti e accessibili a tutti. Algoritmi difficili o inconsueti sono stati descritti con attente spiegazioni degli aspetti matematici. Se il lettore ha già confidenza con un argomento, troverà i capitoli organizzati in modo che possa sorvolare sui paragrafi di introduzione passando velocemente al materiale più avanzato.

Questo libro tratta molti argomenti, che probabilmente non saranno affrontati tutti durante un corso. Pertanto si è cercato di renderlo utile sia come libro di testo sia come manuale di riferimento da usare in seguito nella carriera professionale.

I prerequisisti per poterlo leggere sono:

- una qualche esperienza di programmazione, la comprensione delle procedure ricorsive e la conoscenza di alcune strutture di dati semplici come gli array e le liste;
- una certa confidenza con le dimostrazioni per induzione matematica; poche parti del libro si basano su nozioni di analisi matematica e la Parte I del libro presenta tutte le tecniche matematiche necessarie.

Al professionista

La varietà degli argomenti di questo libro lo rende un ottimo manuale sugli algoritmi. Poiché ogni capitolo è relativamente completo, il lettore può focalizzare la sua attenzione sugli argomenti che maggiormente lo interessano.

Poiché molti degli algoritmi discussi hanno grande utilità pratica, ne sono considerati anche gli aspetti realizzativi: si forniscono alternative pratiche anche per quei pochi algoritmi che sono essenzialmente di interesse teorico.

Per il lettore che desideri realizzare uno qualsiasi degli algoritmi presentati sarà semplice tradurre lo pseudocodice nel suo linguaggio di programmazione preferito. Lo pseudocodice è stato progettato per presentare ciascun algoritmo in modo chiaro e sintetico; sono stati volutamente ignorati la gestione degli errori ed altri aspetti dell'ingegneria del software che richiedono ipotesi più specifiche sull'ambiente di programmazione. Si è cercato di presentare ciascun algoritmo in modo semplice e diretto, evitando le idiosincrasie di un linguaggio di programmazione specifico che avrebbe potuto confondere il suo significato.

Errori

Un libro di questa mole contiene senz'altro errori ed omissioni. Se il lettore trova qualche errore o ha qualche utile suggerimento da proporre, sarà ascoltato volentieri. Saranno particolarmente apprezzati suggerimenti per nuovi esercizi e problemi, purché includano la soluzione. I commenti possono essere inviati a

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

Si può anche utilizzare la posta elettronica su Internet per far conoscere errori, per richiedere una lista degli errori conosciuti o per proporre suggerimenti costruttivi. Per ricevere informazioni, spedire all'indirizzo elettronico `algorithms@theory.lcs.mit.edu` specificando "`Subject: help`" nell'intestazione del messaggio. Siamo dolenti di non poter rispondere personalmente a tutti i messaggi.

Ringraziamenti

Molti amici e colleghi hanno contribuito considerevolmente alla qualità di questo libro. Li ringraziamo per l'aiuto e le critiche costruttive.

Il Laboratory for Computer Science del MIT ha fornito un ottimo ambiente di lavoro. I nostri colleghi del Theory of Computation Group del laboratorio sono stati particolarmente d'aiuto, tolleranti malgrado le nostre richieste incessanti di verifica e critica dei capitoli. Un particolare ringraziamento va a Baruch Awerbuch, Shafi Goldwasser, Leo Guibas, Tom Leighton, Albert Meyer, David Shmoys, e Eva Tardos. Grazie a William Ang, Sally Bemus, Ray Hirschfeld e Mark Reinhold che hanno garantito il buon funzionamento dei nostri calcolatori (DEC Microvax, Apple Macintosh, e Sun Sparcstation) e la ricompilazione di TEX tutte le volte che si superava il limite del tempo di compilazione. La Thinking Machines Corporation ha contribuito permettendo a Charles Leiserson di lavorare a questo libro durante una sua assenza dall'MIT.

Molti colleghi hanno usato versioni preliminari di questo testo per tenere corsi in altre scuole, ed hanno suggerito numerose correzioni e revisioni. In particolare si ringraziano Richard Beigel (Yale), Andrew Goldberg (Stanford), Joan Lucas (Rutgers), Mark Overmars (Utrecht), Alan Sherman (Tuft e Maryland) e Diane Souvaine (Rutgers).

Molti assistenti dei nostri corsi hanno dato contributi significativi allo sviluppo di questo materiale. Si ringraziano in particolare 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 e Margaret Tuttle.

Molte altre persone hanno fornito un'utile e valida assistenza tecnica. Denise Sergent ha trascorso molte ore nelle biblioteche del MIT alla ricerca di riferimenti bibliografici. Maria Sensale, la bibliotecaria della nostra sala di lettura è stata sempre disponibile e di grande aiuto. L'accesso alla biblioteca personale di Albert Meyer ci ha fatto risparmiare molte ore nella preparazione delle note ai capitoli. Shlomo Kipnis, Bill Niehaus e David Wilson hanno corretto le bozze dei vecchi esercizi, ne hanno sviluppato di nuovi e hanno scritto le note per le loro soluzioni. Marios Papaefthymiou e Gregory Troxel hanno contribuito alla stesura dell'indice analitico. Negli anni, le nostre segretarie Inna Radziohovsky, Denise Sergent, Gayle Sherman e, particolarmente, Be Hubbard hanno fornito un sostegno senza pari per questo progetto, e pertanto le ringraziamo.

Molti errori nelle prime versioni sono stati individuati dagli studenti. Un particolare ringraziamento va a Bobby Blumofe, Bonnie Eisenberg, Raymond Johnson, John Keen, Richard Lethin, Mark Lillibridge, John Pezaris, Steve Ponzio e Margaret Tuttle per la loro attenta lettura.

Molti colleghi hanno anche fornito revisioni critiche di capitoli specifici, o informazioni su algoritmi specifici, pertanto siamo loro grati. In particolare si ringraziano Bill Aiello, Alok Aggarwal, Eric Bach, Vasek Chvátal, Richard Cole, Johan Hastad, Alex Ishii, David

Johnson, Joe Kilian, Dina Kravets, Bruce Maggs, Jim Orlin, James Park, Thane Plamback, Hershel Safer, Jeff Shallit, Cliff Stein, Gil Strang, Bob Tarjan e Paul Wang. Alcuni colleghi hanno anche gentilmente suggerito alcuni problemi; si ringraziano in modo particolare Andrew Goldberg, Danny Sleator e Umesh Vazirani.

Il libro originale è stato impaginato con LaTeX, un insieme di macro per TeX. Le figure sono state realizzate su Apple Macintosh usando MacDraw II; si ringraziano Joanna Terry della Claris Corporation e Michael Mahoney della Advanced Computer Graphics per la loro disponibilità. L'indice analitico è stato compilato usando Windex, un programma C scritto dagli autori. La bibliografia è stata preparata usando BibTeX. Il libro è stato impaginato presso la American Mathematical Society con un fotocompositore Autologic; si ringrazia Ralph Youngen della AMS per il suo aiuto. La copertina del libro è stata progettata da Jeannet Leendertse. L'estetica dell'impaginazione del libro è stato curata da Rebecca Daw, e Amy Hendrickson l'ha realizzata in LaTeX.

È stato un piacere lavorare con MIT Press e McGraw-Hill per la realizzazione di questo libro. Si ringraziano in particolare Frank Satlow, Terry Ehling, Larry Cohen e Lorrie Lejeune di MIT Press, e David Shapiro di McGraw-Hill per il loro incoraggiamento, aiuto e pazienza.

Siamo particolarmente grati a Larry Cohen per il suo notevole lavoro di redazione.

Infine, ringraziamo le nostre mogli – Nicole Cormen, Linda Lue Leiserson e Gail Rivest – e i nostri figli – Ricky, William e Debby Leiserson e Alex e Christopher Rivest – per il loro amore e il loro aiuto durante la scrittura di questo libro. (Alex Rivest ci ha anche aiutato per il "paradosso del compleanno del marziano".) L'amore, la pazienza e l'incoraggiamento delle nostre famiglie hanno reso possibile questo progetto. Affettuosamente dedichiamo loro questo libro.

Cambridge, Massachusetts

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

Introduzione

1

In questo capitolo il lettore potrà prendere confidenza con l'organizzazione che adotteremo in tutto il libro per presentare il progetto e l'analisi di algoritmi. Il capitolo è indipendente, tuttavia include alcuni riferimenti a materiale che sarà più dettagliatamente analizzato nei capitoli successivi della Parte I.

Cominceremo con una discussione generale sui problemi computazionali e sugli algoritmi necessari alla loro risoluzione, utilizzando il problema dell'ordinamento come esempio guida. Si introduce anche uno "pseudocodice", che risulterà più familiare ai lettori che hanno già studiato linguaggi di programmazione, utile per mostrare una possibile versione degli algoritmi proposti. L'*insertion sort*, che è un semplice algoritmo di ordinamento, servirà come esempio iniziale di utilizzo di "pseudocodice". Si analizzerà il tempo di esecuzione di questo algoritmo, introducendo una notazione che metta in luce come tale tempo aumenti in funzione del numero degli elementi da ordinare. Verrà presentato anche l'approccio *divide-et-impera* per la progettazione di algoritmi; tale approccio sarà usato per sviluppare un algoritmo chiamato *merge sort*. Il capitolo termina con il confronto tra questi due algoritmi di ordinamento.

1.1 Algoritmi

Informalmente, un ***algoritmo*** è una qualsiasi procedura computazionale ben definita che prende alcuni valori, o un insieme di valori, come ***input*** e produce alcuni valori, o un insieme di valori, come ***output***. Un algoritmo è quindi una sequenza di passi computazionali che trasformano l'input nell'output.

Un algoritmo può anche essere considerato come uno strumento per risolvere un ben definito ***problema computazionale***: infatti la definizione del problema specifica, in termini generali, la relazione che deve valere tra input ed output e l'algoritmo descrive una procedura computazionale specifica per raggiungere tale relazione tra input ed output.

Si comincerà lo studio degli algoritmi con il problema di ordinare una sequenza di numeri in ordine non decrescente. Questo problema si presenta frequentemente nella pratica e fornisce un fertile terreno per introdurre molti strumenti di analisi e tecniche standard di progettazione. Formalmente, il ***problema dell'ordinamento*** si definisce nel seguente modo:

Input: Una sequenza di n numeri $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: Una permutazione (riordinamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ della sequenza di input tale che $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Data la sequenza di input ⟨31, 41, 59, 26, 41, 58⟩, un algoritmo di ordinamento restituisce come output la sequenza ⟨26, 31, 41, 41, 58, 59⟩. Una sequenza di input di questo tipo è chiamata un'istanza del problema di ordinamento. Più in generale, un'***istanza di un problema*** consiste di tutti i dati di ingresso (che soddisfino i vincoli imposti nella definizione del problema) necessari a calcolare una soluzione.

L'ordinamento è un'operazione fondamentale in informatica (in molti programmi è usato come passo intermedio), e infatti sono stati sviluppati un gran numero di algoritmi di ordinamento. La scelta del migliore algoritmo per una data applicazione dipende dal numero di elementi da ordinare, da quanto gli elementi siano già di per sé ordinati in qualche modo e dal tipo di dispositivo di memoria da usare: memoria principale, dischi o nastri.

Un algoritmo si dice ***corretto*** se, per ogni istanza di input, si ferma con l'output corretto; si dice che un algoritmo corretto ***risolve*** il problema computazionale dato. Un algoritmo scorretto potrebbe non fermarsi per qualche istanza di input, o potrebbe fermarsi con una risposta diversa da quella prevista. Contrariamente a quanto ci si potrebbe aspettare, algoritmi scorretti possono qualche volta essere utili se il loro tasso di errore può essere controllato; un esempio di applicazione di algoritmi di questo tipo si vedrà nel Capitolo 33 quando saranno studiati algoritmi per la ricerca di grandi numeri primi. In generale, comunque, si esamineranno solo algoritmi corretti.

Un algoritmo può essere descritto in linguaggio corrente, come un programma per un calcolatore o anche come un progetto hardware; l'unica condizione che deve valere è che la specifica fornita descriva in modo preciso la procedura computazionale che deve essere seguita.

Nel libro gli algoritmi saranno di solito descritti come programmi scritti in uno ***pseudocodice***, un linguaggio che somiglia molto al C, al Pascal o all'Algol. Il lettore che già conosce questi linguaggi non dovrebbe avere difficoltà nella lettura degli algoritmi; infatti, ciò che distingue lo pseudocodice da un codice "reale" è che nello pseudocodice, per descrivere un dato algoritmo, si impiegano metodi espressivi più chiari e concisi.

Talvolta, il metodo più chiaro è l'uso del linguaggio corrente, per cui non sorprenda la presenza di frasi o periodi in linguaggio naturale, mescolati a "vero" codice. Un'altra differenza tra lo pseudocodice ed un codice reale è che il primo non è specificatamente concepito per gestire aspetti di ingegneria del software: l'astrazione dei dati, la modularità e la gestione degli errori, per esempio, sono spesso ignorati, in modo da cogliere più direttamente l'essenza di un algoritmo.

Insertion sort

Iniziamo con l'algoritmo chiamato ***insertion sort***, che risulta efficiente nel caso si debba ordinare un piccolo numero di elementi. L'insertion sort funziona nello stesso modo usato da molte persone per ordinare una mano di bridge o ramino; si inizia con la mano sinistra vuota e le carte coperte poste sul tavolo, quindi si prende dal tavolo una carta alla volta e la si inserisce nella corretta posizione nella mano sinistra. Per trovare la giusta posizione per una carta la si confronta con ogni altra carta già nella mano, da destra a sinistra, come si vede dalla figura 1.1.

Lo pseudocodice per l'algoritmo di insertion sort è descritto con una procedura chiamata INSERTION-SORT, che prende come parametro un array $A[1 \,..\, n]$ contenente una sequenza di lunghezza n che deve essere ordinata. (Nel codice, il numero n di elementi di A corrisponde a *length*$[A]$). I numeri in input sono ***ordinati in loco***: i numeri sono risistemati dentro l'array

Figura 1.1 *Insertion-Sort su una mano di carte.*

A, con al più un numero costante di loro memorizzati, ad ogni istante, fuori dall'array. Quando la procedura INSERTION-SORT termina, l'array di input A contiene la sequenza ordinata di output.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2      do   key ← A[j]
3           ▷ Si inserisce A[j] nella sequenza ordinata A[1 .. j − 1]
4           i ← j − 1
5           while i > 0 e A[i] > key
6                 do  A[i + 1] ← A[i]
7                     i ← i − 1
8           A[i + 1] ← key
```

La figura 1.2 mostra il funzionamento dell'algoritmo per $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. L'indice j indica la "carta corrente" che deve essere inserita nella mano, gli elementi $A[1 \,..\, j-1]$ dell'array costituiscono le carte già ordinate che sono nella mano ed infine, gli elementi $A[j+1 \,..\, n]$ corrispondono al mazzetto di carte ancora sul tavolo.

L'indice j si sposta da sinistra a destra su tutto l'array; ad ogni iterazione del ciclo **for** "più esterno", l'elemento $A[j]$ è copiato fuori dall'array (linea 2), quindi cominciando dalla posizione $j-1$, gli elementi sono spostati ad uno ad uno di una posizione a destra finché non viene trovata la giusta posizione per $A[j]$ (linee 4-7): a questo punto l'elemento $A[j]$ è inserito (linea 8).

Figura 1.2 *L'elaborazione di Insertion-Sort sull'array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. La posizione di indice j è indicata con un cerchio.*

Convenzioni sullo pseudocodice

Nello pseudocodice saranno usate le seguenti convenzioni.

1. Le rientranze indicano la struttura dei blocchi. Per esempio, il corpo del ciclo **for**, che comincia dalla linea 1, comprende le linee 2-8; mentre il corpo del ciclo **while**, che comincia dalla linea 5, contiene le linee 6-7 ma non la linea 8. Le rientranze si applicano nello stesso modo ai costrutti **if-then-else**. L'uso della rientranza al posto di più convenzionali indicatori della struttura di un blocco, come il costrutto **begin-end**, riduce notevolmente la lunghezza della procedura preservandone e talvolta migliorandone la chiarezza[1].
2. I costrutti iterativi **while**, **for** e **repeat** e quelli condizionali **if**, **then** ed **else** hanno la stessa interpretazione del Pascal.
3. Il simbolo "▷" indica che il resto della linea è un *commento*.
4. Un assegnamento multiplo della forma $i \leftarrow j \leftarrow e$ assegna ad entrambe le variabili i e j il valore dell'espressione e; questo assegnamento dovrebbe essere considerato equivalente all'assegnamento $j \leftarrow e$ seguito da $i \leftarrow j$.
5. Le variabili (come i, j e *key*) sono *locali* alla procedura data e comunque non saranno mai usate variabili *globali* senza un'esplicita indicazione.
6. Si può accedere agli elementi di un array specificando il nome dell'array seguito dall'indice scritto tra parentesi quadre. Per esempio $A[i]$ indica lo i-esimo elemento di un array A. La notazione ".." denota un intervallo di valori dentro un array, quindi $A[1 \,..\, j]$ indica il sottoarray di A contenente gli elementi $A[1], A[2], \ldots, A[j]$.
7. I dati composti sono generalmente organizzati in ***oggetti***, che sono strutturati in ***attributi*** o ***campi***. Si può accedere a un campo specifico usando il nome del campo seguito dal suo oggetto scritto tra parentesi quadre. Per esempio, un array è trattato come un oggetto con l'attributo *length* che indica quanti elementi contiene, quindi per specificare quanti elementi ha un array A, si scrive *length*$[A]$. Benché si usino le parentesi quadre sia per l'indicizzazione di un array che per gli attributi di un oggetto, l'interpretazione da utilizzare sarà chiara in base al contesto.

[1] Nei linguaggi di programmazione reali, generalmente non si usa soltanto la rientranza per indicare la struttura di un blocco, in quanto i livelli di rientranza sono complicati da determinare quando il codice è suddiviso su più pagine.

Una variabile che rappresenta un array o un oggetto è trattata come un *puntatore* ai dati contenuti nell'array o nell'oggetto. Per tutti i campi f di un oggetto x, l'assegnamento $y \leftarrow x$ implica che $f[y] = f[x]$. Inoltre, se dopo si esegue l'assegnamento $f[x] \leftarrow 3$, allora non soltanto $f[x] = 3$ ma anche $f[y] = 3$; in altre parole x e y dopo l'assegnamento $y \leftarrow x$, puntano allo stesso oggetto ("sono" lo stesso oggetto).

Talvolta un puntatore non si riferisce a nessun oggetto in particolare, in questo caso gli è assegnato il valore speciale NIL.

8. Ad ogni procedura i parametri sono passati ***per valore***: la procedura chiamata riceve una propria copia dei parametri e se si assegna un valore ad un parametro, la modifica *non* è sentita dalla routine chiamante. Quando si passano oggetti, viene passata una copia del puntatore ai dati che rappresentano l'oggetto, mentre non sono copiati i campi degli oggetti. Per esempio, se x è un parametro di una procedura chiamata, l'assegnamento $x \leftarrow y$ dentro la procedura non è visibile alla procedura chiamante, mentre l'assegnamento $f[x] \leftarrow 3$ sarà visibile.

Esercizi

1.1-1 Usando la figura 1.2 come modello, illustrare l'operazione di INSERTION-SORT applicata all'array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

1.1-2 Riscrivere la procedura INSERTION-SORT per ordinare in ordine non crescente anziché in ordine non decrescente.

1.1-3 Considerare il seguente ***problema di ricerca***:

Input: una sequenza di n numeri $A = \langle a_1, a_2, \ldots, a_n \rangle$ ed un valore v.

Output: un indice i tale che $A[i] = v$, oppure il valore speciale NIL se v non è presente in A. Scrivere lo pseudocodice per la ***ricerca lineare*** che scandisce la sequenza a partire dal primo elemento per ricercare il valore v.

1.1-4 Considerare il problema di addizionare due interi codificati ciascuno su n bit, memorizzati in due array A e B di n elementi. La somma dei due interi dovrebbe essere memorizzata in forma binaria in un array C di $(n + 1)$ elementi. Descrivere il problema in modo formale e scrivere lo pseudocodice per addizionare i due interi.

1.2 Analisi di algoritmi

Analizzare un algoritmo ha il significato di prevedere le risorse che l'algoritmo richiede. Talvolta sono di primaria importanza risorse come la *memoria*, porte di comunicazione e porte logiche; molto più spesso è interessante misurare il *tempo* computazionale. Generalmente, analizzando diversi algoritmi candidati per la soluzione di un problema, il più efficiente può essere identificato facilmente; tali analisi possono indicare più di un candidato, mentre parecchi algoritmi inferiori sono di solito scartati nel corso dell'analisi.

Prima di poter analizzare un algoritmo, si deve stabilire un *modello* della tecnologia di realizzazione che verrà usata, incluso un modello delle risorse per tale tecnologia e dei loro costi. Nella maggior parte del libro, come tecnologia di realizzazione si considererà un generico mono-processore basato sul modello computazionale indicato con ***random-access machine*** (***RAM***). Gli algoritmi saranno effettivamente realizzati come programmi per tale calcolatore. Nel modello RAM le istruzioni sono eseguite una dopo l'altra senza operazioni concorrenti. Negli ultimi capitoli, tuttavia, si avrà occasione di studiare modelli per calcolatori paralleli e circuiti digitali.

L'analisi di un algoritmo può risultare complessa anche se l'algoritmo è semplice; gli strumenti matematici che occorrono possono includere elementi di combinatoria discreta, teoria del calcolo della probabilità e richiedere competenze di algebra e capacità di identificare i termini più significativi di una formula. Dato che il comportamento di un algoritmo può essere diverso per ogni possibile input, è necessario un modo per generalizzare questo comportamento in formule semplici e facilmente comprensibili.

Infine, pur avendo selezionato un solo modello di calcolo per analizzare un dato algoritmo, sarà necessario far comunque fronte a molte scelte nel decidere come esprimere l'analisi: l'obiettivo più immediato è quello di trovare un metodo espressivo che sia semplice da scrivere e manipolare, capace di evidenziare caratteristiche importanti sulle risorse richieste dall'algoritmo e di tralasciare i dettagli irrilevanti.

L'analisi di insertion sort

Il tempo impiegato dalla procedura INSERTION-SORT dipende dall'input: l'ordinamento di un migliaio di numeri richiede più tempo dell'ordinamento di tre numeri. Inoltre la procedura INSERTION-SORT può impiegare quantità diverse di tempo per ordinare sequenze della stessa dimensione, a seconda di quanto queste siano già parzialmente ordinate. In generale, il tempo richiesto da un algoritmo cresce con la dimensione dell'input, per cui è tradizione descrivere il tempo di esecuzione di un programma come una funzione della dimensione del suo input. Per far ciò è necessario descrivere in modo più formale i termini "tempo di esecuzione" e "dimensione dell'input".

La nozione migliore di ***dimensione dell'input*** dipende dal problema che si deve studiare: per molti problemi, come per esempio l'ordinamento o il calcolo delle trasformate discrete di Fourier, la misura più naturale è il *numero degli elementi dell'input* – per esempio, la dimensione n dell'array per l'ordinamento; per molti altri problemi, come per esempio la moltiplicazione di due numeri interi, la misura migliore della dimensione dell'input è il *numero totale di bit* necessari a rappresentare l'input nella consueta rappresentazione binaria. Talvolta, inoltre, è più appropriato descrivere la dimensione dell'input con due numeri piuttosto che con uno solo: per esempio, se l'input di un algoritmo è un grafo, la dimensione dell'input può essere descritta dal numero dei nodi e degli archi presenti nel grafo. In generale, comunque, per ogni problema studiato sarà sempre indicata la misura scelta per la dimensione dell'input.

Il ***tempo di esecuzione*** di un algoritmo per un particolare input è rappresentato dal numero di operazioni elementari o "passi" eseguiti per il calcolo dell'output corrispondente: risulta conveniente definire la nozione di passo nel modo più indipendente possibile da una qualunque macchina. A tale scopo, per il momento, si possono fare le seguenti ipotesi: per eseguire ciascuna linea di pseudocodice è richiesto un tempo costante e una linea può richiedere un tempo di esecuzione diverso da quello di un'altra linea. Si assumerà che ogni

esecuzione della i-esima riga impieghi un tempo c_i, dove c_i è costante. Questa ipotesi è compatibile con il modello RAM e con le realizzazioni effettive dello pseudocodice nella maggior parte dei calcolatori moderni[2].

Nella discussione che segue, l'espressione del tempo di esecuzione per la procedura INSERTION-SORT evolverà da una formula piuttosto confusa che usa tutti i costi c_i, ad una notazione più semplice, più concisa e più facilmente manipolabile. Inoltre questa notazione più semplice permetterà di determinare più facilmente se un algoritmo è più efficiente di un altro.

Inizieremo l'analisi presentando la procedura INSERTION-SORT con il tempo "speso" da ciascuna linea ed il numero di volte che ogni linea è eseguita. Per ciascun $j = 2, 3, \ldots, n$, dove $n = length[A]$, sia t_j il numero di volte che il test del ciclo ***while*** alla linea 5 è eseguito per quel valore di j. Si assume che i commenti non siano eseguiti e perciò non richiedano tempo di esecuzione.

INSERTION-SORT(A)		*costo*	*n° di volte*
1	**for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	▷ Si inserisce $A[j]$ nella ▷ sequenza ordinata $A[1 \ldots j-1]$	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ e $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

Il tempo di esecuzione dell'algoritmo è la somma dei tempi di esecuzione di ciascuna linea eseguita: un'istruzione che impiega c_i passi per essere eseguita ed è elaborata n volte contribuirà con $c_i n$ al totale del tempo di esecuzione[3]. Per calcolare $T(n)$, il tempo di esecuzione di INSERTION-SORT, si sommano i prodotti delle colonne dei *costi* e delle *volte*, ottenendo

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

Anche per input di una stessa dimensione, il tempo di esecuzione di un algoritmo può dipendere da *quale tipo* di input è dato. Per esempio, per INSERTION-SORT, il caso migliore è quando l'array è già ordinato: per ciascun $j = 2, 3, \ldots, n$, si trova che $A[i] \leq key$ nella linea 5, quando i ha il suo valore iniziale di $j-1$.

[2] Ci sono alcuni particolari che devono essere sottolineati. I passi computazionali specificati in linguaggio corrente sono spesso varianti di procedure che richiedono un tempo di esecuzione non costante: ad esempio, più avanti si dirà "ordinare i punti in funzione della coordinata x", che, come si vedrà, richiede più che un tempo di esecuzione costante. Inoltre, si noti che la chiamata ad un sottoprogramma impiega un tempo costante anche se il sottoprogramma, una volta invocato, può impiegare un tempo maggiore; cioè si separa il processo della ***chiamata*** al sottoprogramma – passaggio dei parametri etc. – dal processo di ***esecuzione*** del sottoprogramma.

[3] Questa caratteristica non vale necessariamente per una risorsa, come la memoria; un comando che fa riferimento ad m parole di memoria ed è eseguito n volte non necessariamente consuma in totale mn parole di memoria.

Quindi $t_j = 1$ per $j = 2, 3, \ldots, n$, con un tempo di esecuzione nel *caso migliore* pari a

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

Quest'ultimo tempo di esecuzione può essere espresso come $an + b$ per qualche costante a e b dipendente dai costi c_i; quindi $T(n)$ è una ***funzione lineare*** di n.

Se l'array è ordinato in ordine inverso – cioè in *ordine decrescente* – allora si ha il *caso peggiore*. Infatti bisogna confrontare ogni elemento $A[j]$ con ogni elemento nell'intero sottoarray $A[1 \ldots j-1]$, così che $t_j = j$ per $j = 2, 3, \ldots, n$. Notando che

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

e

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

(queste sommatorie saranno riesaminate nel Capitolo 3), si trova che nel caso peggiore il tempo di esecuzione di INSERTION-SORT è

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

Questo tempo di esecuzione del caso peggiore può essere espresso come $an^2 + bn + c$ per costanti a, b e c che dipendono ancora dai costi c_i dei comandi; risulta quindi una ***funzione quadratica*** di n.

Generalmente, come nel caso dell'insertion sort, il tempo di esecuzione di un algoritmo è fisso per un dato input, nonostante ciò negli ultimi capitoli si vedranno alcuni interessanti algoritmi "randomizzati" il cui comportamento può variare anche su un input fissato.

Analisi del caso peggiore e del caso medio

Nell'analisi dell'algoritmo di insertion sort, sono stati presi in considerazione sia il caso migliore, in cui l'array di input è già ordinato, che il caso peggiore, quello in cui l'array di input è ordinato in modo inverso.

Nel seguito del libro sarà di solito preso in considerazione soltanto il ***tempo di esecuzione nel caso peggiore***, cioè il tempo di esecuzione più lungo per qualsiasi input di dimensione n; questa scelta è motivata dalle seguenti tre ragioni.

- Il tempo di esecuzione nel caso peggiore di un algoritmo rappresenta una limitazione superiore sui tempi di esecuzione per qualsiasi input. La sua conoscenza garantisce che l'algoritmo non impiegherà mai tempi maggiori; inoltre non c'è la necessità di studiare nessun'altra ipotesi sul fatto che il tempo di esecuzione possa peggiorare.

- Per alcuni algoritmi, il caso peggiore si verifica abbastanza di frequente. Per esempio, nella ricerca di una particolare informazione in una base di dati, il caso peggiore dell'algoritmo di ricerca si verificherà tutte le volte che quell'informazione non è presente nella base di dati; ed in alcune applicazioni relative alla ricerca di informazioni, la ricerca per dati assenti può essere frequente.
- Il "caso medio" è, in genere, circa altrettanto cattivo quanto il caso peggiore. Si supponga infatti di aver scelto n numeri a caso a cui applicare l'insertion sort; quanto tempo si impiega per determinare dove inserire $A[j]$ nel sottoarray $A[1 \ldots j-1]$? Mediamente metà elementi di $A[1 \ldots j-1]$ sono più piccoli di $A[j]$ e quelli dell'altra metà sono più grandi.

  In media, perciò, si deve verificare solo una metà di $A[1 \ldots j-1]$ e quindi $t_j = j/2$. Calcolando il risultante tempo di esecuzione nel caso medio, si ottiene una funzione quadratica della dimensione dell'input, proprio come per il tempo di esecuzione nel caso peggiore.

Il tempo di esecuzione nel caso ***medio***, detto anche tempo di esecuzione ***atteso***, sarà esaminato in alcuni casi particolari; rimane tuttavia il problema di riconoscere ciò che è un input "medio" per un certo problema. Spesso si assumerà che tutti gli input della stessa dimensione siano equamente probabili; nella pratica questa ipotesi può essere violata, tuttavia gli *algoritmi randomizzati* possono talvolta rendere vera, forzatamente, tale ipotesi.

Ordine di grandezza

Per facilitare l'analisi della procedura INSERTION-SORT si sono fatte alcune *astrazioni* che hanno semplificato lo studio. Innanzitutto si è ignorato l'effettivo costo di ciascun comando, usando le costanti c_i per rappresentare questi costi; inoltre, si è osservato che anche queste costanti forniscono più dettagli di quanto si necessiti: il tempo di esecuzione nel caso peggiore è $an^2 + bn + c$ per costanti a, b e c che dipendono dai costi c_i dei comandi. Quindi non solo si sono ignorati i costi reali dei comandi, ma anche i loro costi astratti c_i.

Si può fare un'ulteriore operazione di astrazione considerando il ***tasso di crescita*** o ***ordine di grandezza*** del tempo di esecuzione, che poi è il valore di maggior interesse nell'analisi degli algoritmi. A tale scopo, si considera solo il termine principale di una formula (per esempio an^2), dato che i termini di ordine inferiore sono relativamente non significativi per valori di n molto grandi; si può ignorare anche il coefficiente costante del termine principale, dato che i fattori costanti sono meno significativi del tasso di crescita nella determinazione dell'efficienza computazionale per input grandi. Quindi, si scrive che l'insertion sort, per esempio, ha un tempo di esecuzione nel caso peggiore di $\Theta(n^2)$ (pronunciato "theta di n al quadrato"). La notazione Θ sarà usata in questo capitolo in modo informale, mentre sarà definita in modo preciso nel Capitolo 2.

Infine, un algoritmo si considera più efficiente di un altro se il suo tempo di esecuzione nel caso peggiore ha un ordine di grandezza inferiore. Questa valutazione può essere errata per input piccoli, ma per input sufficientemente grandi un algoritmo $\Theta(n^2)$, per esempio, sarà eseguito nel caso peggiore più velocemente di un algoritmo $\Theta(n^3)$.

Esercizi

1.2-1 Si consideri il problema di ordinare n numeri, memorizzati in un array A, nel modo seguente: si trova prima il più piccolo elemento di A e si mette nel primo elemento

di un altro array B; poi si trova il secondo più piccolo elemento di A e si mette nel secondo elemento di B e si continua così per tutti gli n elementi di A. Scrivere lo pseudocodice di questo algoritmo conosciuto come ***selection sort***. Fornire i tempi di esecuzione nel caso migliore e nel caso peggiore usando la notazione Θ.

1.2-2 Si consideri ancora la ricerca lineare (si veda l'Esercizio 1.1-3). Quanti elementi della sequenza di input è necessario verificare in media, ipotizzando che l'elemento da cercare possa essere uguale, con la stessa probabilità, ad uno qualunque degli elementi nell'array? Quanti nel caso peggiore? Quali sono i tempi di esecuzione nel caso medio e nel caso peggiore in notazione Θ? Giustificare la risposta.

1.2-3 Si consideri il problema di valutare un polinomio in un punto. Dati n coefficienti $a_0, a_1, \ldots, a_{n-1}$ ed un numero reale x, si vuole calcolare $\sum_{i=0}^{n-1} a_i x^i$. Descrivere, per questo problema, un semplice algoritmo che abbia un tempo $\Theta(n^2)$. Descrivere poi un algoritmo con un tempo $\Theta(n)$ che usi il seguente metodo (chiamato *regola di Horner*) per riscrivere il polinomio:

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

1.2-4 Esprimere la funzione $n^3/1000 - 100n^2 - 1000n + 3$ in termini di notazione Θ.

1.2-5 Come può essere modificato, quasi sempre, un algoritmo per avere un buon tempo di esecuzione nel caso migliore?

1.3 Progetto di algoritmi

Ci sono molte tecniche per progettare algoritmi; per esempio l'algoritmo di insertion sort usa un approccio ***incrementale***: dopo avere ordinato il sottoarray $A[1 \,.\,.\, j-1]$, si inserisce il singolo elemento $A[j]$ nel proprio posto, producendo il sottoarray $A[1 \,.\,.\, j]$ ordinato.

In questo paragrafo si esaminerà un approccio di progettazione alternativo, conosciuto come "divide-et-impera", che sarà impiegato per progettare un algoritmo di ordinamento il cui tempo di esecuzione nel caso peggiore risulta assai inferiore a quello dell'insertion sort.

Un vantaggio degli algoritmi divide-et-impera è che i loro tempi di esecuzione possono essere spesso facilmente determinati usando delle tecniche che saranno esaminate nel Capitolo 4.

1.3.1 L'approccio divide-et-impera

Molti utili algoritmi hanno una struttura ***ricorsiva***: per risolvere un determinato problema, tali algoritmi richiamano se stessi per gestire sottoproblemi analoghi a quello dato. Questi algoritmi generalmente seguono l'approccio ***divide-et-impera***: sezionano il problema in diversi sottoproblemi che sono simili a quello originale ma di dimensione inferiore; risolvono i sottoproblemi ricorsivamente e quindi combinano queste soluzioni per creare una soluzione del problema di partenza.

Il paradigma divide-et-impera, a ciascun livello di ricorsione, si struttura in tre passi:

Divide: il problema è suddiviso in un certo numero di sottoproblemi.

Impera: i sottoproblemi sono risolti ricorsivamente. Tuttavia, se la dimensione dei sottoproblemi è piccola a sufficienza, essi sono risolti direttamente.

Combina: le soluzioni dei sottoproblemi sono combinati per ottenere la soluzione del problema originale.

L'algoritmo ***merge sort*** segue in modo stretto il paradigma divide-et-impera; intuitivamente esso funziona nel modo seguente:

Divide: divide gli n elementi della sequenza da ordinare in due sottosequenze di $n/2$ elementi ciascuna.

Impera: ordina, usando ricorsivamente il merge sort, le due sottosequenze.

Combina: fonde le due sottosequenze per produrre come risposta la sequenza ordinata.

Si noti che il processo delle chiamate ricorsive "risale" quando la sequenza da ordinare ha una lunghezza uguale ad 1: in questo caso non c'è altro da fare in quanto ogni sequenza di lunghezza 1 è di per sé ordinata.

L'operazione chiave dell'algoritmo di merge sort è la ***fusione*** delle due sottosequenze ordinate nel passo "combina". Per eseguire una tale fusione si usa una procedura ausiliaria MERGE (A, p, q, r), dove A è un array, p, q ed r sono indici di elementi dell'array tali che $p \le q < r$; questa procedura, assumendo $A[p \,.\,.\, q]$ e $A[q+1 \,.\,.\, r]$ ordinati, genera un singolo sottoarray che sostituisce il corrente sottoarray $A[p \,.\,.\, r]$.

Benché il corrispondente pseudocodice sia lasciato come esercizio (si veda l'Esercizio 1.3-2), si può facilmente immaginare una procedura MERGE che impieghi un tempo $\Theta(n)$, dove $n = r - p + 1$ è il numero degli elementi su cui si esegue la fusione. Infatti, tornando all'esempio del giocatore di carte, si può supporre di avere sul tavolo due mazzetti di carte scoperte. Ciascun mazzetto è ordinato ed ha in cima la carta più piccola e si desidera fondere i due mazzetti in un unico mazzetto di output. Il passo base consiste nello scegliere la più piccola tra le due carte che si trovano in cima ai due mazzetti, rimuoverla dal suo mazzetto (lasciando così esposta una nuova carta) e deporla coperta in cima al mazzetto di output. Questo passo si ripete finché un mazzetto di input è vuoto, a quel punto si prendono le carte rimanenti dell'altro mazzetto e si pongono, coperte, in cima al mazzetto di output. Computazionalmente, ciascun passo base impiega un tempo costante, in quanto si stanno considerando solo le due carte in cima ai due mazzetti, per cui, al più, si eseguiranno n passi base e quindi la fusione impiega un tempo $\Theta(n)$.

Si può usare adesso la procedura MERGE come un sottoprogramma dell'algoritmo merge sort. La procedura MERGE-SORT (A, p, r) ordina gli elementi del sottoarray $A[p \,.\,.\, r]$; se $p \ge r$ allora il sottoarray ha al più un elemento ed è perciò già ordinato; altrimenti, il passo di divisione calcola semplicemente un indice q che partiziona $A[p \,.\,.\, r]$ in due sottoarray: $A[p \,.\,.\, q]$ contenente $\lceil n/2 \rceil$ elementi e $A[q+1 \,.\,.\, r]$ contenente $\lfloor n/2 \rfloor$ elementi[1].

[1] L'espressione $\lceil x \rceil$ denota il più piccolo intero maggiore o uguale ad x e $\lfloor x \rfloor$ denota il più grande intero minore o uguale ad x. Queste notazioni sono definite nel Capitolo 2.

Figura 1.3 *L'elaborazione di merge-sort sull'array A = ⟨5, 2, 4, 6, 1, 3, 2, 6⟩. La lunghezza delle sequenze ordinate su cui fare la fusione aumenta via via che l'algoritmo progredisce dal basso all'alto.*

```
MERGE-SORT (A, p, r)
if p < r
    then q ← ⌊(p + r)/2⌋
         MERGE-SORT (A, p, q)
         MERGE-SORT (A, q + 1, r)
         MERGE (A, p, q, r)
```

Per ordinare l'intera sequenza $A = \langle A[1], A[2], \ldots, A[n] \rangle$, si chiama la procedura MERGE-SORT(A, 1, *length*[A]), dove ancora una volta *length*[A] = n. Se si guarda al funzionamento bottom up della procedura quando n è una potenza di due, l'algoritmo esegue la fusione di coppie di sequenze di un elemento per formare sequenze ordinate di lunghezza due; poi, esegue la fusione di sequenze di lunghezza 2 per formare sequenze ordinate di lunghezza 4 e così via finché si esegue la fusione su due sottosequenze di lunghezza $n/2$ per formare la sequenza ordinata di lunghezza n. Questo processo è mostrato nella figura 1.3.

1.3.2 Analisi di algoritmi divide-et-impera

Quando un algoritmo contiene una chiamata ricorsiva a se stesso, il suo tempo di esecuzione può spesso essere descritto con una ***equazione di ricorrenza*** o ***ricorrenza***, che descrive il tempo di esecuzione complessivo di un problema di dimensione n in termini del tempo di esecuzione per input più piccoli. Si possono quindi usare strumenti matematici per risolvere la ricorrenza e così fornire limitazioni alle prestazioni di un algoritmo.

Una ricorrenza per il tempo di esecuzione di un algoritmo divide-et-impera è basata sui tre passi del paradigma di base. Come prima, sia $T(n)$ il tempo di esecuzione di un problema di dimensione n; se la dimensione del problema è sufficientemente piccola, per esempio $n \le c$ per qualche costante c, allora la risoluzione banalmente impiega un tempo costante, che si scrive come $\Theta(1)$. Si supponga di suddividere il problema in a sottoproblemi, ciascuno dei quali ha una dimensione $1/b$ rispetto al problema principale; se si impiega un tempo $D(n)$ per *dividere* il problema nei sottoproblemi ed un tempo $C(n)$ per *combinare* le soluzioni dei sottoproblemi nella soluzione del problema originale, si ottiene la seguente ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{altrimenti}\,. \end{cases}$$

Nel Capitolo 4 si vedrà come risolvere ricorrenze ordinarie di questa forma.

Analisi del merge sort

Benché lo pseudocodice di MERGE-SORT funzioni correttamente anche quando il numero degli elementi non è pari, l'analisi basata sulla ricorrenza è semplificata se si ipotizza che il problema originale abbia una dimensione che è una potenza di due. In tal caso, ciascun passo di divisione produce due sottosequenze grandi esattamente $n/2$; nel Capitolo 4 si vedrà che questa ipotesi non influenza l'ordine di grandezza della soluzione della ricorrenza.

Per determinare la ricorrenza $T(n)$ del tempo di esecuzione nel caso peggiore del merge sort su n numeri, si può ragionare nel seguente modo. Il merge sort applicato ad un singolo elemento impiega un tempo costante; quando si hanno $n > 1$ elementi, si suddivide il tempo di esecuzione come mostrato di seguito.

Divide: il passo di divisione calcola l'indice di mezzo dell'array impiegando, quindi, un tempo costante, da cui $D(n) = \Theta(1)$.

Impera: si risolvono ricorsivamente due sottoproblemi, ciascuno di dimensione $n/2$, che contribuiscono per $2T(n/2)$ al tempo di esecuzione.

Combina: si è già esaminato il fatto che la procedura MERGE su sottoarray di n elementi impiega un tempo $\Theta(n)$, per cui $C(n) = \Theta(n)$.

Quando si addizionano le funzioni $C(n)$ e $D(n)$ per l'analisi del merge sort, si stanno addizionando una funzione che è $\Theta(n)$ ed una funzione che è $\Theta(1)$. Questa somma è una funzione lineare di n, cioè $\Theta(n)$; addizionandola al termine del passo "impera", si ottiene la ricorrenza $T(n)$ per il tempo di esecuzione nel caso peggiore del merge sort:

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

Nel Capitolo 4 si vedrà che $T(n)$ è $\Theta(n \lg n)$, dove $\lg n$ sta per $\log_2 n$. Per input sufficientemente grandi e considerando il caso peggiore, il merge sort, con un tempo di esecuzione $\Theta(n \lg n)$, risulta migliore di insertion sort il cui tempo di esecuzione è $\Theta(n^2)$.

Esercizi

1.3-1 Usando la figura 1.3 come modello, eseguire le operazioni del merge sort sull'array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

1.3-2 Scrivere un programma in pseudocodice per MERGE (A, p, q, r).

1.3-3 Usare l'induzione matematica per mostrare che, quando n è una potenza esatta di 2, la soluzione della ricorrenza

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

è $T(n) = n \lg n$.

1.3-4 L'insertion sort può essere espresso come una procedura ricorsiva nel seguente modo: per ordinare $A[1 \,..\, n]$, si ordina ricorsivamente $A[1 \,..\, n-1]$ e quindi si inserisce $A[n]$ nell'array ordinato $A[1 \,..\, n-1]$. Scrivere una ricorrenza per il tempo di esecuzione di questa versione ricorsiva di insertion sort.

1.3-5 Facendo riferimento al *problema della ricerca* (si veda l'Esercizio 1.1-3), si può osservare che se la sequenza A è ordinata, si può confrontare v con l'elemento che occupa la posizione centrale di A ed eliminare da ulteriori confronti metà della sequenza. La ***ricerca binaria*** è un algoritmo che ripete questa procedura, dimezzando ad ogni passo la dimensione della restante porzione della sequenza. Scrivere lo pseudocodice, iterativo o ricorsivo, per la ricerca binaria. Mostrare che il tempo di esecuzione della ricerca binaria nel caso peggiore è $\Theta(\lg n)$.

1.3-6 Osservare che il ciclo **while** nelle linee 5-7 della procedura INSERTION-SORT del paragrafo 1.1 usa una ricerca lineare per scandire (all'indietro) il sottoarray ordinato $A[1 \,..\, j-1]$. Per migliorare il tempo di esecuzione complessivo del caso peggiore dell'insertion sort fino a $\Theta(n \lg n)$, si può usare invece una ricerca binaria (si veda l'Esercizio 1.3-5)?

* *1.3-7* Descrivere un algoritmo con tempo $\Theta(n \lg n)$ che, dato un insieme S di n numeri reali e un altro numero reale x, determini se esistono due elementi di S la cui somma sia esattamente x.

1.4 Riepilogo

Un buon algoritmo è come un coltello affilato: fa esattamente ciò che si suppone debba fare applicando una quantità minima di forza. Invece usare un algoritmo sbagliato per risolvere un problema è come tagliare una bistecca con un cacciavite: si può anche arrivare ad un risultato accettabile, ma facendo sicuramente uno sforzo più grande di quanto non fosse necessario, e inoltre il risultato non sarà certo elegante.

Algoritmi ideati per risolvere lo stesso problema spesso differiscono vistosamente per quanto riguarda la loro efficienza e queste differenze possono essere più significative delle differenze tra un personal computer ed un supercalcolatore: si può vedere, per esempio, come un supercalcolatore su cui viene eseguito l'algoritmo di insertion sort, possa essere affossato da un personal computer su cui viene eseguito il merge sort.

Si supponga che i due algoritmi debbano ordinare un array di un milione di numeri, che il supercalcolatore esegua 100 milioni di istruzioni al secondo mentre il personal computer sia in grado di eseguire soltanto un milione di istruzioni al secondo.

Per rendere la differenza ancora più vistosa si supponga anche che il più furbo programmatore del mondo abbia codificato l'insertion sort nel linguaggio macchina del supercalcolatore ed il codice risultante richieda $2n^2$ istruzioni del supercalcolatore per ordinare n numeri; viceversa, si supponga che il merge sort sia stato programmato per il personal computer da un programmatore medio usando un linguaggio ad alto livello che dispone di un compilatore non efficiente, ottenendo un codice che richiede $50n \lg n$ istruzioni del personal computer.

Per ordinare un milione di numeri il supercalcolatore impiega

$$\frac{2 \cdot (10^6)^2 \text{ istruzioni}}{10^8 \text{ istruzioni/secondo}} = 20000 \text{ secondi} \approx 5.56 \text{ ore},$$

mentre il personal computer impiega

$$\frac{50 \cdot 10^6 \cdot \lg 10^6 \text{ istruzioni}}{10^6 \text{ istruzioni/secondo}} \approx 1000 \text{ secondi} \approx 16.67 \text{ minuti}.$$

Quindi, usando un algoritmo il cui tempo di esecuzione ha un ordine di grandezza più piccolo, anche con un compilatore scadente, il personal computer gira 20 volte più velocemente rispetto al supercalcolatore!

Questo esempio mostra che gli algoritmi, come l'hardware, sono ***tecnologia*** e la prestazione globale di un sistema dipende tanto dalla scelta di algoritmi efficienti quanto dalla scelta di un hardware veloce. Così come sono stati fatti rapidi progressi in altre tecnologie dei calcolatori, così ve ne sono stati nel campo della tecnologia degli algoritmi.

Esercizi

1.4-1 Si supponga di voler confrontare su una stessa macchina insertion sort e merge sort. Per input di dimensione n, l'insertion sort viene eseguito in $8n^2$ passi, mentre il merge sort viene eseguito in $64n \lg n$ passi. Per quali valori di n l'insertion sort è più conveniente del merge sort? Come si potrebbe riscrivere lo pseudocodice del merge sort per renderlo più efficiente anche su input piccoli?

1.4-2 Qual è il più piccolo valore di n tale che, su una stessa macchina, un algoritmo il cui tempo di esecuzione è di $100n^2$ venga eseguito più velocemente di un algoritmo il cui tempo di esecuzione è di 2^n?

1.4-3 Si consideri il problema di determinare se una sequenza arbitraria $\langle x_1, x_2, \ldots, x_n \rangle$ di n numeri contenga occorrenze ripetute dello stesso numero. Mostrare che questo può essere fatto in tempo $\Theta(n \lg n)$.

Problemi

1-1 *Confronto fra tempi di esecuzione*

Per ogni funzione $f(n)$ e tempo t della tabella seguente, determinare il più grande input n di un problema che può essere risolto in un tempo t, assumendo che l'algoritmo per risolvere tale problema impieghi $f(n)$ microsecondi.

$f(n)$ \ t	1 secondo	1 minuto	1 ora	1 giorno	1 mese	1 anno	1 secolo
$\lg n$							
$\sqrt{n}$							
$n \lg n$							
n^2							
n^3							
2^n							
$n!$							

1-2 *Insertion sort nel merge sort su array piccoli*

Benché merge sort venga eseguito nel caso peggiore in tempo $\Theta(n \lg n)$ ed insertion sort impieghi un tempo $\Theta(n^2)$ sempre considerando il caso peggiore, i fattori costanti di insertion sort lo rendono più veloce per valori piccoli di n. Quindi ha senso usare insertion sort dentro il merge sort quando i sottoproblemi diventano sufficientemente piccoli. Si consideri una modifica del merge sort in cui n/k sottoliste di lunghezza k sono ordinate usando l'insertion sort e sono poi combinate usando il meccanismo standard di fusione, con k che deve essere determinato.

a. Mostrare che le n/k sottoliste, ciascuna di lunghezza k, nel caso peggiore possono essere ordinate dall'insertion sort con un tempo $\Theta(nk)$.

b. Mostrare che le sottoliste possono essere fuse in tempo $\Theta(n \lg(n/k))$, sempre considerando il caso peggiore.

c. Dato che l'algoritmo modificato viene eseguito, nel caso peggiore, in un tempo $\Theta(nk + n \lg(n/k))$, qual è il più grande valore asintotico (notazione Θ) di k come funzione di n, per cui l'algoritmo modificato ha lo stesso tempo di esecuzione asintotico del merge sort standard?

d. Come, nella pratica, dovrebbe essere scelto il valore di k?

1-3 *Inversioni*

Sia $A[1 \ldots n]$ un array di n numeri distinti. Se $i < j$ e $A[i] > A[j]$, allora la coppia (i, j) è chiamata ***inversione*** di A.

a. Elencare le cinque inversioni dell'array $\langle 2, 3, 8, 6, 1 \rangle$.

b. Quale array con elementi presi dall'insieme $\{1, 2, \ldots, n\}$ ha più inversioni? Quante sono le inversioni di questo array?

c. Qual è la relazione tra il tempo di esecuzione dell'insertion sort ed il numero di inversioni dell'array di input? Giustificare la risposta.

d. Scrivere un algoritmo che determini il numero di inversioni di una qualunque permutazione di n elementi in tempo $\Theta(n \lg n)$ nel caso peggiore. (*Suggerimento*: Modificare il merge sort).

Note al capitolo

Vi sono molti buoni libri che affrontano in generale l'argomento degli algoritmi, inclusi quelli di Aho, Hopcroft e Ullman [4, 5]; Baase [14]; Brassard e Bratley [33]; Horowitz e Sahni [105]; Knuth[121, 122, 123]; Manber [142]; Mehlhorn[144, 145, 146]; Purdom e Brown [164]; Reingold, Nievergelte Deo [167]; Sedgewick [175]; Wil [201]. In Bentley [24, 25] e Gonnet [90] sono discussi alcuni degli aspetti più pratici nella progettazione degli algoritmi.

Nel 1968 Knuth pubblicò il primo di tre volumi con il titolo generale di "The Art of Computer Programming" [121, 122, 123], che avviò lo studio moderno degli algoritmi richiamando l'attenzione sull'analisi del tempo di esecuzione; tutta la serie, comunque, rimane un utile ed importante riferimento per molti degli argomenti di questo libro. Secondo Knuth, la parola "algoritmo" deriva dal nome "al-Khowârizmî", un matematico persiano del IX secolo.

Aho, Hopcroft e Ullman [4] mostrarono come l'analisi asintotica degli algoritmi sia uno strumento di confronto tra prestazioni relative. Essi diffusero anche l'uso delle relazioni di ricorrenza per descrivere il tempo di esecuzione degli algoritmi ricorsivi.

Knuth [123] fornisce un trattato enciclopedico di molti algoritmi di ordinamento: il suo confronto di tali algoritmi (pagina 381) include l'analisi esatta del conteggio dei passi, come quella che è stata fatta nel libro per l'insertion sort. La discussione di Knuth sull'algoritmo di insertion sort comprende diverse varianti dell'algoritmo, la più importante di queste è lo Shell sort, presentato da D. L. Shell, che usa insertion sort su sottosequenze periodiche dell'input ottenendo un algoritmo di ordinamento più veloce.

Il merge sort è descritto anche da Knuth, che cita un selezionatore meccanico, inventato nel 1938, capace di eseguire la fusione di due pacchi di schede perforate in un solo passo. Sembra che anche J. von Neumann, uno dei pionieri dell'informatica, abbia scritto nel 1945 un programma di merge sort per il calcolatore EDVAC.

I Fondamenti di matematica

Introduzione

L'analisi degli algoritmi spesso richiede l'impiego di numerosi strumenti matematici; alcuni di essi sono semplici come l'algebra delle scuole superiori, ma altri, come quelli per la soluzione di ricorrenze, potrebbero risultare al lettore delle vere novità. Questa parte del libro vuol essere proprio un compendio dei metodi e degli strumenti che si useranno per analizzare algoritmi: essa è organizzata in primo luogo come un testo di riferimento e talvolta come introduzione ad alcuni argomenti particolarmente rilevanti.

Si suggerisce al lettore di non provare a leggere tutti questi fondamenti matematici in una sola volta. Conviene, piuttosto, sfogliare prima i capitoli di questa parte per vedere gli argomenti contenuti e poi procedere direttamente ai capitoli che riguardano gli algoritmi. Questa parte può sempre essere utilizzata come riferimento per una migliore comprensione degli strumenti usati nel libro per l'analisi degli algoritmi.

Il Capitolo 2 presenta le definizioni di diverse notazioni asintotiche, un esempio delle quali è la notazione Θ che si è già incontrata nel Capitolo 1. Il resto del capitolo è essenzialmente basato sulla presentazione di nozioni matematiche e mira a far sì che risulti chiara la notazione che sarà usata nel resto del libro.

Il Capitolo 3 riguarda i metodi per la valutazione e la limitazione delle sommatorie, che si presentano spesso nell'analisi degli algoritmi. Anche se molte delle formule di questo capitolo si possono trovare in un qualsiasi libro di matematica, sarà utile averle raccolte tutte insieme per riferimenti successivi.

I metodi per la risoluzione delle ricorrenze, che sono state usate nel Capitolo 1 per analizzare il merge sort, sono raccolti nel Capitolo 4; una tecnica potente è il "metodo generale", che si usa per risolvere le ricorrenze che si trovano analizzando gli algoritmi del tipo "divide-et-impera". La maggior parte del capitolo è dedicata alla dimostrazione di correttezza di questo metodo, che però può essere saltata senza alcun danno per la comprensione successiva degli argomenti proposti.

Il Capitolo 5 contiene le definizioni di base e le notazioni per gli insiemi, le relazioni, le funzioni, i grafi e gli alberi; inoltre fornisce le proprietà di base di questi oggetti matematici. Questo materiale è essenziale per la comprensione del libro, ma può essere tranquillamente saltato dal lettore che abbia già seguito altri corsi su questi argomenti.

Il Capitolo 6 comincia con i principi elementari del calcolo: permutazioni, combinazioni e simili. Il resto del capitolo contiene le definizioni e le proprietà di base del calcolo delle probabilità. La maggior parte degli algoritmi di questo libro non richiede il calcolo delle

probabilità per la loro analisi, quindi, ad una prima lettura, gli ultimi paragrafi del capitolo possono essere saltati anche senza sfogliarli. Tuttavia, se si vuole comprendere meglio l'analisi probabilistica che si incontrerà in seguito, il Capitolo 6 è organizzato in modo da essere un comodo riferimento.

Ordine di grandezza delle funzioni

2

L'ordine di grandezza del tempo di esecuzione di un algoritmo, definito nel Capitolo 1, fornisce una semplice caratterizzazione dell'efficienza di un algoritmo e consente anche di confrontare le prestazioni di algoritmi alternativi. Quando la dimensione n dell'input diventa sufficientemente grande, il merge sort, con il suo tempo di esecuzione $\Theta(n \lg n)$ nel caso peggiore, risulta migliore dell'insertion sort, il cui tempo di esecuzione nel caso peggiore è $\Theta(n^2)$. Sebbene si possa talvolta determinare il tempo esatto di esecuzione di un algoritmo, come si è fatto per l'insertion sort nel Capitolo 1, l'estrema precisione non è di solito così importante da compensare lo sforzo del calcolo; infatti per input sufficientemente grandi, le costanti moltiplicative e i termini di ordine più basso dell'esatto tempo di esecuzione sono trascurabili rispetto agli effetti della dimensione stessa dell'input.

Quando si considerano input sufficientemente grandi da rendere rilevante solo l'ordine di grandezza del tempo di esecuzione, si sta studiando l'efficienza ***asintotica*** degli algoritmi. In tal caso ciò che interessa è come cresce il tempo di esecuzione di un algoritmo al crescere all'infinito della dimensione dell'input. Di solito, un algoritmo che sia asintoticamente più efficiente costituisce la scelta migliore per tutti gli input tranne quelli veramente piccoli.

Questo capitolo fornisce alcuni metodi standard per semplificare l'analisi degli algoritmi. Il prossimo paragrafo comincia con la definizione di alcuni tipi di "notazione asintotica", di cui si è già visto un esempio nella notazione Θ; quindi saranno presentate alcune convenzioni notazionali usate nel libro e infine sarà fornita una panoramica sul comportamento delle funzioni che di solito si presentano nell'analisi degli algoritmi.

2.1 Notazione asintotica

Le notazioni usate per descrivere il tempo asintotico di esecuzione di un algoritmo sono definite in termini di funzioni il cui dominio è l'insieme dei numeri naturali $\mathbb{N} = \{0, 1, 2, \ldots\}$. Queste notazioni sono comode per descrivere la funzione $T(n)$ del tempo di esecuzione nel caso peggiore, definita di solito solo su dimensioni intere dell'input. Talvolta è comunque conveniente *usare in modo improprio* la notazione asintotica. Per esempio la notazione è facilmente estesa al dominio dei numeri reali o, alternativamente, ristretta a un sottoinsieme dei numeri naturali. È importante però comprendere il significato preciso della notazione in modo che anche quando sia usata impropriamente, non sia usata male. Questo paragrafo definisce la notazione asintotica di base e introduce inoltre alcuni comuni usi impropri.

Notazione Θ

Nel Capitolo 1 si è determinato che il tempo di esecuzione nel caso peggiore dell'algoritmo insertion sort è $T(n) = \Theta(n^2)$. Questa notazione ha il seguente significato.

Data una funzione $g(n)$, si denota con $\Theta(g(n))$ l'*insieme di funzioni*

$$\Theta(g(n)) = \{ f(n): \text{ esistono tre costanti positive } c_1, c_2 \text{ e } n_0 \text{ tali che } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ per ogni } n \ge n_0 \}.$$

Una funzione $f(n)$ appartiene all'insieme $\Theta(g(n))$ se esistono le costanti positive c_1 e c_2 tali che essa possa essere schiacciata tra $c_1 g(n)$ e $c_2 g(n)$ per n sufficientemente grande. Malgrado $\Theta(g(n))$ sia un insieme, per indicare che $f(n)$ è membro di $\Theta(g(n))$ si scrive "$f(n) = \Theta(g(n))$", oppure si scrive "$f(n) \in \Theta(g(n))$". Questo uso improprio del segno di uguaglianza per denotare l'appartenenza insiemistica, può sembrare, almeno all'inizio, confusionario, ma si vedrà che ha i suoi vantaggi.

La figura 2.1 (a) fornisce una rappresentazione intuitiva delle funzioni $f(n)$ e $g(n)$, dove $f(n) = \Theta(g(n))$. Per ogni valore di n alla destra di n_0, il valore di $f(n)$ coincide o si trova sopra $c_1 g(n)$ e coincide o si trova sotto $c_2 g(n)$. In altre parole, per ogni $n \ge n_0$, la funzione $f(n)$ è uguale a $g(n)$ a meno di un fattore costante. Si dice che $g(n)$ è un ***limite asintotico stretto*** per $f(n)$.

La definizione di $\Theta(g(n))$ richiede che ogni membro $f(n)$ di $\Theta(g(n))$ sia ***asintoticamente non negativo***, cioè che $f(n)$ sia non negativa tutte le volte che n è sufficientemente grande. Di conseguenza, la stessa funzione $g(n)$ deve essere asintoticamente non negativa, altrimenti l'insieme $\Theta(g(n))$ è vuoto. Si assumerà perciò che ogni funzione usata dentro la notazione Θ sia asintoticamente non negativa; questa ipotesi vale anche per le altre notazioni asintotiche definite in questo capitolo. Si dice ***asintoticamente positiva*** una funzione che sia strettamente positiva per tutti gli n sufficientemente grandi.

Nel Capitolo 1 si è introdotta una nozione informale della notazione Θ che consisteva nel tralasciare i termini di grado più basso ed ignorare il coefficiente principale del termine di ordine più grande. Si può brevemente giustificare questa soluzione intuitiva usando la

Figura 2.1 *Esempi grafici delle notazioni Θ, O, e Ω. In ogni parte della figura, il valore di n_0 mostrato è il minimo valore possibile; qualunque valore più grande va bene.* (a) *La notazione Θ limita una funzione rispetto a fattori costanti. Si scrive $f(n) = \Theta(g(n))$ se esistono delle costanti positive n_0, c_1 e c_2 tali che alla destra di n_0 il valore di $f(n)$ si trova sempre tra $c_1 g(n)$ e $c_2 g(n)$ inclusi.* (b) *La notazione O fornisce un limite superiore per una funzione rispetto a fattori costanti. Si scrive $f(n) = O(g(n))$ se esistono delle costanti positive n_0 e c tali che alla destra di n_0 il valore di $f(n)$ è sempre sotto o coincide con $cg(n)$.* (c) *La notazione Ω fornisce un limite inferiore per una funzione rispetto a fattori costanti. Si scrive $f(n) = \Omega(g(n))$ se esistono delle costanti positive n_0 e c tali che alla destra di n_0 il valore di $f(n)$ è sempre sopra o coincide con $cg(n)$.*

definizione formale per mostrare che $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Per far questo si devono determinare le costanti positive c_1, c_2 ed n_0 tali che

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

per ogni $n \ge n_0$. Dividendo per n^2 si ha

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

La disuguaglianza della parte destra può essere resa valida per qualunque valore di $n \ge 1$, scegliendo $c_2 \ge 1/2$. Analogamente, la disuguaglianza della parte destra può essere resa valida per qualunque valore di $n \ge 7$, scegliendo $c_1 \le 1/14$. Quindi, scegliendo $c_1 = 1/14$, $c_2 = 1/2$ ed $n_0 = 7$, si può verificare che $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certamente esistono altre scelte per le costanti, ma la cosa importante è che qualche scelta esista. Si noti che queste costanti dipendono dalla funzione $\frac{1}{2}n^2 - 3n$; una diversa funzione appartenente a $\Theta(n^2)$ di solito richiederà costanti diverse.

Si può usare la definizione formale anche per verificare che $6n^3 \ne \Theta(n^2)$. Si supponga per assurdo che esistano c_2 ed n_0 tali che $6n^3 \le c_2 n^2$ per ogni $n \ge n_0$. Allora $n \le c_2/6$, che non può valere in alcun modo per n comunque grande, dato che c_2 è costante.

Intuitivamente, i termini di ordine più basso di una funzione asintoticamente positiva possono essere ignorati nella determinazione del limite asintotico stretto perché essi sono trascurabili per n grande. Una frazione piccola del termine di ordine più alto è sufficiente a dominare i termini di ordine più basso.

Quindi l'assegnamento a c_1 di un valore leggermente più piccolo del coefficiente del termine di ordine più alto e a c_2 di un valore leggermente più grande, consente che le disuguaglianze nella definizione della notazione Θ siano soddisfatte. Il coefficiente del termine più alto può analogamente essere ignorato, in quanto esso cambia c_1 e c_2 solo per un fattore costante uguale al coefficiente.

Come esempio si consideri una generica funzione quadratica $f(n) = an^2 + bn + c$, dove a, b e c sono costanti ed $a > 0$. Tralasciando i termini di ordine più basso ed ignorando la costante si ha $f(n) = \Theta(n^2)$. Formalmente, per mostrare la stessa cosa, si prendono le costanti $c_1 = a/4$, $c_2 = 7a/4$ ed $n_0 = 2\max((|b|/a), \sqrt{(|c|/a)})$. Il lettore può verificare che $c_1 n^2 \le an^2 + bn + c \le c_2 n^2$ per ogni $n \ge n_0$. In generale per qualunque polinomio $p(n) = \sum_{i=0}^{d} a_i n^i$, dove le a_i sono costanti e $a_d > 0$, si ha $p(n) = \Theta(n^d)$ (si veda il Problema 2-1).

Dato che qualunque costante è un polinomio di grado 0, si può esprimere qualunque funzione costante come $\Theta(n^0)$ o $\Theta(1)$. Quest'ultima notazione, poiché non è chiaro quale variabile stia tendendo all'infinito, è un uso improprio, comunque di minore importanza[1]. Si userà spesso la notazione $\Theta(1)$ per indicare o una costante o una funzione costante rispetto a qualche variabile.

Notazione *O*

La notazione Θ limita asintoticamente una funzione da sopra e da sotto. Quando si ha solo un ***limite asintotico superiore***, si usa la notazione *O*. Per una funzione data $g(n)$, si denota con $O(g(n))$ l'insieme di funzioni

[1] Il problema reale è che la nostra notazione ordinaria per le funzioni non fa distinzione tra funzioni e valori. Nel λ-calcolo, i parametri delle funzioni sono chiaramente specificati: la funzione n^2 potrebbe essere scritta come $\lambda n.n^2$ o anche $\lambda r.r^2$. D'altra parte adottando una notazione più rigorosa si potrebbe complicare la manipolazione algebrica, e così si preferisce tollerare quest'uso improprio.

$O(g(n)) = \{f(n)$: esistono c ed n_0 tali che $0 \le f(n) \le cg(n)$ per ogni $n \ge n_0\}$.

La notazione O si usa per dare un limite superiore ad una funzione a meno di un fattore costante. La figura 2.1 (b) mostra il concetto intuitivo che sta dietro la notazione O: per tutti i valori di n alla destra di n_0 il valore della funzione $f(n)$ coincide o si trova sotto $g(n)$.

Per indicare che una funzione $f(n)$ è membro di $O(g(n))$, si scrive $f(n) = O(g(n))$; si noti che $f(n) = \Theta(g(n))$ implica che $f(n) = O(g(n))$ in quanto la notazione Θ è una notazione più forte della notazione O; per dirlo con la teoria degli insiemi $\Theta(g(n)) \subseteq O(g(n))$. Di conseguenza la dimostrazione che qualunque funzione quadratica $an^2 + bn + c$, dove $a > 0$, è in $\Theta(n^2)$ implica anche che ogni funzione quadratica è in $O(n^2)$. Vale la pena notare che qualunque funzione *lineare* $an + b$ è in $O(n^2)$, il che è facilmente verificabile prendendo $c = a + |b|$ ed $n_0 = 1$.

Qualche lettore che avesse già visto la notazione O può trovare strano che si possa scrivere, per esempio, $n = O(n^2)$. In letteratura, la notazione O è usata talvolta per descrivere informalmente limiti asintoticamente stretti, cioè quelli che sono stati qui definiti usando la notazione Θ.

In questo libro, comunque, quando si scrive $f(n) = O(g(n))$, si sta semplicemente affermando che qualche funzione $g(n)$ è un limite asintotico di $f(n)$ senza affermare quanto il limite superiore sia stretto; la distinzione tra limiti asintotici superiori e limiti asintoticamente stretti è ormai diventata standard nella letteratura riguardante gli algoritmi.

Usando la notazione O si può spesso descrivere il tempo di esecuzione di un algoritmo semplicemente analizzando la struttura complessiva dell'algoritmo; per esempio, per la struttura ciclica doppiamente annidata dell'algoritmo di insertion sort del Capitolo 1, si ha un limite superiore $O(n^2)$ del tempo di esecuzione nel caso peggiore: il costo del ciclo più interno è limitato superiormente da $O(1)$ (costante), gli indici i e j valgono entrambi al più n, e il ciclo più interno è eseguito al più una volta per ognuna delle n^2 coppie di valori di i e j.

Poiché la notazione O descrive un limite superiore, quando la si usa per limitare il tempo di esecuzione di un algoritmo nel caso peggiore, per implicazione si limita anche il tempo di esecuzione dell' algoritmo su input arbitrari. Per cui il limite $O(n^2)$ del tempo di esecuzione di insertion sort nel caso peggiore si applica anche al tempo di esecuzione su qualunque input.

Viceversa, il limite $\Theta(n^2)$ del tempo di esecuzione di insertion sort nel caso peggiore non implica un limite $\Theta(n^2)$ al tempo di esecuzione su qualunque input. Per esempio nel Capitolo 1 si è visto che quando l'input è già ordinato, l'insertion sort viene eseguito in tempo $\Theta(n)$.

Tecnicamente, è un uso improprio dire che il tempo di esecuzione di insertion sort è $O(n^2)$, in quanto, per un dato n, il tempo di esecuzione reale dipende dal particolare input di dimensione n. Cioè, il tempo di esecuzione non è in realtà una funzione solo di n. Ciò che si intende quando si dice "il tempo di esecuzione è $O(n^2)$" è che il tempo di esecuzione nel caso peggiore (che è una funzione di n) è $O(n^2)$ o equivalentemente, qualunque sia il particolare input di dimensione n scelto per ogni valore di n, il tempo di esecuzione su quell'insieme di input è $O(n^2)$.

Notazione Ω

Così come la notazione O fornisce un limite asintotico superiore ad una funzione, la notazione Ω fornisce un ***limite asintotico inferiore***. Per una funzione data $g(n)$, si denota con $\Omega(g(n))$ l'insieme di funzioni

$\Omega(g(n)) = \{f(n)$: esistono c ed n_0 tali che $0 \le cg(n) \le f(n)$ per ogni $n \ge n_0\}$.

Il concetto intuitivo che sta dietro la notazione Ω è mostrato nella figura 2.1(c): per tutti i valori di n alla destra di n_0 il valore della funzione $f(n)$ coincide o si trova sopra $cg(n)$.

Dalle definizioni delle notazioni asintotiche che si sono viste fino adesso, è semplice dimostrare il seguente importante teorema (si veda l'Esercizio 2.1-5).

Teorema 2.1

Per ogni coppia di funzioni $f(n)$ e $g(n)$, $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Come esempio di applicazione di questo teorema, la dimostrazione che $an^2 + bn + c = \Theta(n^2)$ per qualunque costante a, b e c, con $a > 0$, implica immediatamente che $an^2 + bn + c = \Omega(n^2)$ e $an^2 + bn + c = O(n^2)$. In pratica, piuttosto che usare il Teorema 2.1 per ottenere limiti asintotici superiori e inferiori dai limiti asintoticamente stretti, come è stato fatto per quest'esempio, generalmente lo si usa per dimostrare limiti asintoticamente stretti partendo da limiti asintotici superiori ed inferiori.

Dato che la notazione Ω descrive un limite inferiore, quando la si usa per limitare il tempo di esecuzione nel caso migliore di un algoritmo, per implicazione si limita anche il tempo di esecuzione dell'algoritmo su input arbitrari. Per esempio il tempo di esecuzione nel caso migliore dell'algoritmo insertion sort è $\Omega(n)$ che implica che il tempo di esecuzione dell'insertion sort è $\Omega(n)$.

Il tempo di esecuzione dell'insertion sort è perciò compreso tra $\Omega(n)$ e $O(n^2)$, in quanto cade ovunque tra una funzione lineare di n ed una funzione quadratica di n. Inoltre, questi limiti sono il più possibile asintoticamente stretti: per esempio, il tempo di esecuzione dell'insertion sort non è $\Omega(n^2)$ in quanto l'insertion sort viene eseguito in tempo $O(n)$ quando l'input è già ordinato. Non è contraddittorio, tuttavia, dire che il tempo di esecuzione dell'insertion sort nel *caso peggiore* è $\Omega(n^2)$, in quanto esiste un input per il quale l'algoritmo impiega un tempo $\Omega(n^2)$. Quando si dice che il *tempo di esecuzione* (senza ulteriore specificazione) di un algoritmo è $\Omega(g(n))$, si intende che, *a prescindere da quale particolare input di dimensione n sia scelto per ciascun valore di n*, il tempo di esecuzione su quell'insieme di input è almeno una costante moltiplicata per $g(n)$, con n sufficientemente grande.

Notazione asintotica nelle equazioni

Si è già visto come la notazione asintotica possa essere usata all'interno di formule matematiche: per esempio, nell'introdurre la notazione O, si è scritto "$n = O(n^2)$". Si potrebbe anche scrivere $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$. Come interpretare questo tipo di formule?

Quando la notazione asintotica si trova da sola sul lato destro di un'equazione, come in $n = O(n^2)$, il segno uguale, come si è già detto, indica appartenenza: $n \in O(n^2)$. In generale, però, quando la notazione asintotica appare in una formula, la si interpreta come una qualche anonima funzione che non interessa specificare. Per esempio, la formula $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ significa che $2n^2 + 3n + 1 = 2n^2 + f(n)$, dove $f(n)$ è una funzione nell'insieme $\Theta(n)$. In questo caso, $f(n) = 3n + 1$, che è proprio in $\Theta(n)$.

Usando la notazione asintotica in questo modo si possono eliminare da un'equazione confusioni e dettagli inessenziali. Per esempio nel Capitolo 1 il tempo di esecuzione del merge sort nel caso peggiore è stato espresso con la ricorrenza

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

Se si è interessati solo al comportamento asintotico di $T(n)$, non è di alcuna importanza specificare esattamente tutti i termini di ordine più basso; è sottinteso che siano tutti inclusi nella funzione anonima denotata dal termine $\Theta(n)$.

Il numero di funzioni anonime in un'espressione è sottinteso che sia uguale al numero di volte che la notazione asintotica appare. Per esempio, nell'espressione

$$\sum_{i=0}^{n} O(i) ,$$

vi è solo una singola funzione anonima (una funzione di i). Questa espressione quindi *non è* la stessa cosa di $O(1) + O(2) + \ldots + O(n)$, che in realtà non ha una chiara interpretazione.

In alcuni casi, la notazione asintotica appare sul lato sinistro di un'equazione, come in

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

Si interpretano equazioni di questo tipo usando la seguente regola: *indipendentemente da come sono scelte le funzioni anonime nella parte sinistra del segno uguale, vi è un modo per scegliere le funzioni anonime sulla destra del segno uguale per rendere valida l'equazione.* Di conseguenza il significato dell'esempio è che per *qualunque* funzione $f(n) \in \Theta(n)$ vi è una *qualche* funzione $g(n) \in \Theta(n^2)$ tale che $2n^2 + f(n) = g(n)$ per ogni n. In altre parole, il lato destro di un'equazione fornisce un livello di dettaglio più grossolano del lato sinistro.

Varie relazioni di questo tipo possono essere utilizzate insieme a catena come in

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

Si può interpretare ogni equazione separatamente attraverso la regola descritta sopra. La prima equazione dice che vi è *qualche* funzione $f(n) \in \Theta(n)$ tale che $2n^2 + 3n + 1 = 2n^2 + f(n)$ per ogni n. La seconda equazione dice che per *qualsiasi* funzione $g(n) \in \Theta(n)$ (come la $f(n)$ già menzionata) vi è *qualche* funzione $h(n) \in \Theta(n^2)$ tale che $2n^2 + g(n) = h(n)$ per ogni n. Si noti che questa interpretazione implica che $2n^2 + 3n + 1 = \Theta(n^2)$, il che è ciò che la catena di equazioni intuitivamente fornisce.

Notazione o

Il limite asintotico superiore fornito dalla notazione O può essere asintoticamente stretto oppure no; il limite $2n^2 = O(n^2)$ è asintoticamente stretto, mentre il limite $2n = O(n^2)$ non lo è. Si usa la notazione o per denotare un limite superiore che non è asintoticamente stretto. Si definisce formalmente $o(g(n))$ ("o piccolo di g di n") l'insieme

$o(g(n)) = \{f(n)$: per qualunque costante positiva $c > 0$, esiste una costante $n_0 > 0$ tale che $0 \le f(n) < cg(n)$ per ogni $n \ge n_0\}$.

Per esempio, $2n = o(n^2)$, ma $2n^2 \ne o(n^2)$.

Le definizioni della notazione O e della notazione o sono simili; la principale differenza è che in $f(n) = O(g(n))$, il limite $0 \le f(n) \le cg(n)$ vale per *qualche* costante c, mentre in $f(n) = o(g(n))$, il limite $0 \le f(n) < cg(n)$ vale per *tutte* le costanti $c > 0$. Il concetto intuitivo nella notazione o è che la funzione $f(n)$ diventa trascurabile rispetto a $g(n)$ per n che tende all'infinito; cioè,

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

Alcuni autori usano questo limite come definizione della notazione o; la definizione di questo libro limita le funzioni anonime ad essere asintoticamente non negative.

Notazione ω

Per analogia, la notazione ω sta alla notazione Ω come la notazione o sta alla notazione O. Si usa la notazione ω per denotare un limite inferiore non asintoticamente stretto. Un modo per definirla è

$f(n) \in \omega(g(n))$ se e solo se $g(n) \in o(f(n))$.

Formalmente si definisce $\omega(g(n))$ ("omega piccolo di g di n") come l'insieme

$\omega(g(n)) = \{f(n)$: per qualunque costante positiva $c > 0$, esiste una costante $n_0 > 0$ tale che $0 \le cg(n) < f(n)$ per ogni $n \ge n_0\}$.

Per esempio, $n^2/2 = \omega(n)$, ma $n^2/2 \ne \omega(n^2)$. La relazione $f(n) = \omega(g(n))$ implica che

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

se il limite esiste; cioè $f(n)$ diventa arbitrariamente grande rispetto a $g(n)$ per n che tende all'infinito.

Confronto tra funzioni

Molte delle proprietà relazionali dei numeri reali sono valide anche per i confronti tra notazioni asintotiche. Nel seguito si assumerà che $f(n)$ e $g(n)$ siano asintoticamente positive.

Transitività:

$$\begin{aligned}
f(n) &= \Theta(g(n)) \quad \text{e} \quad g(n) = \Theta(h(n)) \text{ implicano } f(n) = \Theta(h(n)) , \\
f(n) &= O(g(n)) \quad \text{e} \quad g(n) = O(h(n)) \text{ implicano } f(n) = O(h(n)) , \\
f(n) &= \Omega(g(n)) \quad \text{e} \quad g(n) = \Omega(h(n)) \text{ implicano } f(n) = \Omega(h(n)) , \\
f(n) &= o(g(n)) \quad \text{e} \quad g(n) = o(h(n)) \text{ implicano } f(n) = o(h(n)) , \\
f(n) &= \omega(g(n)) \quad \text{e} \quad g(n) = \omega(h(n)) \text{ implicano } f(n) = \omega(h(n)) .
\end{aligned}$$

Riflessività:

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

Simmetria:

$$f(n) = \Theta(g(n)) \quad \text{se e solo se} \quad g(n) = \Theta(f(n)) .$$

Simmetria trasposta:

$$\begin{aligned}
f(n) &= O(g(n)) \quad \text{se e solo se} \quad g(n) = \Omega(f(n)) , \\
f(n) &= o(g(n)) \quad \text{se e solo se} \quad g(n) = \omega(f(n)) .
\end{aligned}$$

Poiché queste proprietà valgono per le notazioni asintotiche, si può tracciare la seguente analogia tra il confronto asintotico di due funzioni f e g ed il confronto di due numeri reali a e b:

$$f(n) = O(g(n)) \Rightarrow a \le b,$$
$$f(n) = \Omega(g(n)) \Rightarrow a \ge b,$$
$$f(n) = \Theta(g(n)) \Rightarrow a = b,$$
$$f(n) = o(g(n)) \Rightarrow a < b,$$
$$f(n) = \omega(g(n)) \Rightarrow a > b.$$

Una proprietà dei numeri reali che però non può essere trasferita alla notazione asintotica è la seguente:

Tricotomia: Per ogni coppia di numeri reali a e b, deve valere esattamente una delle seguenti espressioni: $a < b$, $a = b$, $a > b$.

Sebbene qualunque coppia di numeri possa essere confrontata, non tutte le funzioni sono asintoticamente confrontabili. Cioè, per una coppia di *funzioni* $f(n)$ e $g(n)$, può essere che non valga né $f(n) = O(g(n))$ né $f(n) = \Omega(g(n))$. Per esempio le funzioni n e $n^{1+\sin n}$, usando le notazioni asintotiche, non possono essere confrontate in quanto il valore dell'esponente di $n^{1+\sin n}$ oscilla tra 0 e 2, assumendo tutti i valori dell'intervallo.

Esercizi

2.1-1 Siano $f(n)$ e $g(n)$ funzioni asintoticamente non negative. Usando la definizione di base della notazione Θ, dimostrare che $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

2.1-2 Mostrare che per ogni coppia di costanti reali a e b, dove $b > 0$,

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

2.1-3 Spiegare perché l'affermazione: "Il tempo di esecuzione dell'algoritmo A è almeno $O(n^2)$" è priva di significato.

2.1-4 È vero che $2^{n+1} = O(2^n)$? È vero che $2^{2n} = O(2^n)$?

2.1-5 Dimostrare il Teorema 2.1.

2.1-6 Dimostrare che il tempo di esecuzione di un algoritmo è $\Theta(g(n))$ se e solo se il suo tempo di esecuzione nel caso peggiore è $O(g(n))$ e il suo tempo di esecuzione nel caso migliore è $\Omega(g(n))$.

2.1-7 Dimostrare che $o(g(n)) \cap \omega(g(n))$ è l'insieme vuoto.

2.1-8 Si può estendere la notazione al caso di due parametri n ed m che possono tendere indipendentemente all'infinito con tassi di crescita diversi. Per una data funzione $g(n, m)$, si denota con $O(g(n, m))$ l'insieme di funzioni:

$$O(g(n, m)) = \{f(n, m): \text{esistono tre costanti positive } c, n_0 \text{ e } m_0 \text{ tali che } 0 \le f(n, m) \le cg(n, m) \text{ per ogni } n \ge n_0 \text{ e } m \ge m_0\}.$$

Dare le corrispondenti definizioni per $\Omega(g(n, m))$ e $\Theta(g(n, m))$.

2.2 Notazioni standard e funzioni comuni

Questo paragrafo fornisce una panoramica su alcune funzioni e notazioni matematiche standard ed analizza le relazioni tra esse; illustra inoltre l'uso delle notazioni asintotiche.

Monotonicità

Una funzione $f(n)$ è ***monotona crescente*** se $m \le n$ implica $f(m) \le f(n)$; analogamente è ***monotona decrescente*** se $m \le n$ implica $f(m) \ge f(n)$. Una funzione $f(n)$ è ***strettamente crescente*** se $m < n$ implica $f(m) < f(n)$ ed è ***strettamente decrescente*** se $m < n$ implica $f(m) > f(n)$.

Base e tetto

Dato un qualunque numero reale x, si denota il più grande intero minore o uguale a x con $\lfloor x \rfloor$ (si legga "base di x") e il più piccolo intero maggiore o uguale a x con $\lceil x \rceil$ (si legga "tetto di x"). Per tutti i reali x

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

Per qualunque intero n,

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

e per qualunque intero n, a e b interi, con $a \ne 0$ e $b > 0$,

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

e

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

Le funzioni base e tetto sono monotone crescenti.

Polinomi

Dato un intero positivo d, un ***polinomio in n di grado d*** è una funzione $p(n)$ della forma

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

dove le costanti $a_0, a_1, \ldots, a_d$ sono i ***coefficienti*** del polinomio e $a_d \ne 0$. Un polinomio è ***asintoticamente positivo*** se e solo se $a_d > 0$. Dato un polinomio asintoticamente positivo $p(n)$ di grado d, si ha $p(n) = \Theta(n^d)$. Per qualsiasi costante reale $a \ge 0$, la funzione n^a è monotona crescente e per qualsiasi costante reale $a \le 0$, la funzione n^a è monotona decrescente.

Si dice che una funzione $f(n)$ è ***limitata polinomialmente*** se $f(n) = n^{O(1)}$, che è equivalente a dire che $f(n) = O(n^k)$ per qualche costante k (si veda l'Esercizio 2.2-2).

Esponenziali

Per ogni reale $a \ne 0$, m ed n, si hanno le seguenti identità:

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

Per ogni n e $a \geq 1$, la funzione a^n è monotona crescente rispetto a n. Quando sarà conveniente, si assumerà $0^0 = 1$.

Il tasso di crescita di polinomi ed esponenziali può essere messo in relazione con il seguente fatto: per tutte le costanti reali a e b tali che $a > 1$,

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

da cui si può concludere che

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

Quindi, qualunque funzione esponenziale positiva con base strettamente maggiore di 1 cresce più velocemente di qualunque polinomio.

Usando e per denotare 2.71828.... la base della funzione logaritmo naturale, si ha per ogni reale x

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

dove "!" denota la funzione fattoriale definita nel seguito di questo paragrafo. Per ogni reale x, si ha la disuguaglianza

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

dove l' uguaglianza vale solo quando $x = 0$. Quando $|x| \leq 1$, abbiamo l'approssimazione

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

Quando $x \to 0$, l'approssimazione di e^x con $1 + x$ è abbastanza buona:

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

(In questa equazione, la notazione asintotica è usata per descrivere il comportamento al limite per $x \to 0$ piuttosto che per $x \to \infty$.) Si ha che per ogni x:

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

Logaritmi

Si useranno le seguenti notazioni:

$$
\begin{aligned}
\lg n &= \log_2 n && \text{(logaritmo binario).} \\
\ln n &= \log_e n && \text{(logaritmo naturale).} \\
\lg^k n &= (\lg n)^k && \text{(elevamento a potenza).} \\
\lg \lg n &= \lg(\lg n) && \text{(composizione).}
\end{aligned}
$$

Un'importante notazione convenzionale che sarà adottata è che le *funzioni logaritmiche si applicano solo al termine successivo nella formula*, così $\lg n + k$ significherà $(\lg n) + k$ e non $\lg(n + k)$. Per $n > 0$ e $b > 1$, la funzione $\log_b n$ è strettamente crescente.

Per tutti i reali $a > 0$, $b > 0$, $c > 0$ e n, purché $b \neq 1$ e $c \neq 1$, vale che

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

Dato che cambiando la base di un logaritmo da una costante ad un'altra il valore del logaritmo cambia solo per un fattore costante, si userà spesso la notazione "$\lg n$" quando non si sia interessati ai fattori costanti, come nella notazione O. Gli informatici trovano che la base più naturale per i logaritmi sia 2 perché molti algoritmi e strutture dati prevedono la divisione in due parti del problema.

Vi è un semplice sviluppo in serie per $\ln(1 + x)$ quando $|x| < 1$:

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

Per $x > -1$ si hanno anche le seguenti disuguaglianze:

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

dove l'uguaglianza vale solo per $x = 0$.

Si dice che una funzione $f(n)$ è ***polilogaritmicamente limitata*** se $f(n) = \lg^{O(1)} n$. Si può correlare la crescita dei polinomi e dei polilogaritmi sostituendo nell'Equazione (2.5) $\lg n$ al posto di n e 2^a al posto di a ottenendo

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

Da questo limite si può concludere che

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

per qualunque costante $a > 0$. Di conseguenza, qualunque funzione polinomiale positiva cresce più velocemente di qualunque funzione polilogaritmica.

Fattoriali

La notazione $n!$ (si legge "n fattoriale") è definita per interi $n \geq 0$ come

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

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

Un limite superiore debole per la funzione fattoriale è $n! \le n^n$, dato che ciascuno degli n termini del prodotto fattoriale è al più n. L'*approssimazione di Stirling*

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

dove e è la base dei logaritmi naturali, fornisce sia un limite superiore che un limite inferiore più stretto. Usando l'approssimazione di Stirling si può dimostrare che

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

Per ogni n valgono anche i seguenti limiti:

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

La funzione logaritmica iterata

Si usa la notazione $\lg^* n$ (si legge "log stella di n") per denotare l'iterazione logaritmica che è definita come segue:

sia la funzione $\lg^{(i)} n$ definita ricorsivamente per interi non negativi i come

$$\lg^{(i)} n = \begin{cases} n & \text{se } i = 0 , \\ \lg(\lg^{(i-1)} n) & \text{se } i > 0 \text{ e } \lg^{(i-1)} n > 0 , \\ \text{indefinito} & \text{se } i > 0 \text{ e } \lg^{(i-1)} n \le 0 \text{ o } \lg^{(i-1)} n \text{ è indefinito} . \end{cases}$$

Attenzione a distinguere $\lg^{(i)} n$ (la funzione logaritmica applicata i volte in successione partendo con argomento n) da $\lg^i n$ (la funzione logaritmica di n, elevata alla i-esima potenza). La funzione logaritmica iterata è definita come

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

Il logaritmo iterato è una funzione che cresce *molto* lentamente:

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

Poiché il numero di atomi nell'universo osservabile è stimato essere circa 10^{80}, che è molto meno di 2^{65536}, si incontra raramente un valore di n tale che $\lg^* n > 5$.

Numeri di Fibonacci

I *numeri di Fibonacci* sono definiti dalla seguente ricorrenza:

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{per } i \ge 2. \end{aligned} \tag{2.13}$$

Perciò, ogni numero di Fibonacci è la somma dei due precedenti e vale la sequenza

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

I numeri di Fibonacci sono correlati al *rapporto aureo* ϕ e al suo coniugato $\hat{\phi}$ che sono definiti dalle seguenti formule:

$$\begin{aligned} \phi &= \frac{1 + \sqrt{5}}{2} \\ &= 1.61803\ldots, \\ \hat{\phi} &= \frac{1 - \sqrt{5}}{2} \\ &= -0.61803\ldots . \end{aligned} \tag{2.14}$$

Più precisamente, si ha

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

come si può dimostrare per induzione (si veda l'Esercizio 2.2-7). Dato che $|\hat{\phi}| < 1$, si ha che $|\hat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$ e quindi l'i-esimo numero di Fibonacci F_i è uguale a $\phi^i/\sqrt{5}$ arrotondato all'intero più vicino. Perciò i numeri di Fibonacci crescono esponenzialmente.

Esercizi

2.2-1 Mostrare che se $f(n)$ e $g(n)$ sono funzioni monotone crescenti allora lo sono anche le funzioni $f(n) + g(n)$ e $f(g(n))$; inoltre se $f(n)$ e $g(n)$ sono anche non negative allora $f(n) \cdot g(n)$ è monotona crescente.

2.2-2 Usare la definizione della notazione O per mostrare che $T(n) = n^{\lfloor \lg n \rfloor}$ se e solo se esiste una costante $k > 0$ tale che $T(n) = O(n^k)$.

2.2-3 Dimostrare l'Equazione (2.9).

2.2-4 Dimostrare che $\lg(n!) = \Theta(n \lg n)$ e che $n! = o(n^n)$.

* *2.2-5* La funzione $\lceil \lg n \rceil!$ è polinomialmente limitata? La funzione $\lceil \lg\lg n \rceil!$ è polinomialmente limitata?

* *2.2-6* Qual è la funzione asintoticamente più grande: $\lg(\lg^* n)$ o $\lg^*(\lg n)$?

2.2-7 Dimostrare che l'i-esimo numero di Fibonacci soddisfa l'uguaglianza $F_i = (\phi^i - \hat{\phi}^i) / \sqrt{5}$, dove ϕ è il rapporto aureo e $\hat{\phi}$ è il suo coniugato.

2.2-8 Dimostrare che per $i \geq 0$, l'$(i+2)$-esimo numero di Fibonacci soddisfa $F_{i+2} \geq \phi^i$.

Problemi

2-1 *Comportamento asintotico dei polinomi*

Sia

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

dove $a_d > 0$, un polinomio in n di grado d e sia k una costante. Usare le definizioni delle notazioni asintotiche per provare le seguenti proprietà.

a. Se $k \geq d$, allora $p(n) = O(n^k)$.

b. Se $k \leq d$, allora $p(n) = \Omega(n^k)$.

c. Se $k = d$, allora $p(n) = \Theta(n^k)$.

d. Se $k > d$, allora $p(n) = o(n^k)$.

e. Se $k < d$, allora $p(n) = \omega(n^k)$.

2-2 *Crescita asintotica relativa*

Indicare per ogni coppia di espressioni (A, B) nella tabella sotto, se A è O, o, Ω, ω oppure Θ di B. Si assuma che $k \geq 1$, $\varepsilon > 0$ e $c > 1$ siano costanti. La risposta dovrebbe essere sotto forma di tabella con un "sì" o un "no" in ogni casella.

	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 *Ordinamento rispetto al tasso di crescita asintotica*

a. Classificare le seguenti funzioni per ordine di grandezza; cioè, trovare una sistemazione $g_1, g_2, \ldots, g_{30}$ delle funzioni tale che $g_1 = \Omega(g_2)$, $g_2 = \Omega(g_3)$, $\ldots$, $g_{29} = \Omega(g_{30})$.

Partizionare l'elenco in classi di equivalenza tali che $f(n)$ e $g(n)$ sono nella stessa classe se e solo se $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	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. Dare un esempio di una singola funzione non negativa $f(n)$ tale che per ogni funzione $g_i(n)$ della parte (a), $f(n)$ non è né $O(g_i(n))$ né $\Omega(g_i(n))$.

2-4 *Proprietà della notazione asintotica*

Siano $f(n)$ e $g(n)$ due funzioni asintoticamente positive. Provare o confutare ciascuna delle seguenti congetture.

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

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

c. $f(n) = O(g(n))$ implica $\lg(f(n)) = O(\lg(g(n)))$, dove $\lg(g(n)) > 0$ e $f(n) \geq 1$ per ogni n sufficientemente grande.

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

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

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

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

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

2-5 *Variazioni su O e Ω*

Alcuni autori definiscono Ω in modo leggermente diverso da come si faccia qui; si userà $\tilde{\Omega}$ (si legge "omega infinito") per indicare questa definizione alternativa. Si dice che $f(n) = \tilde{\Omega}(g(n))$ se esiste una costante positiva c tale che $f(n) \geq cg(n) \geq 0$ per un numero infinitamente grande di interi n.

a. Mostrare che per qualunque coppia di funzioni $f(n)$ e $g(n)$, asintoticamente non negative, o $f(n) = O(g(n))$ o $f(n) = \tilde{\Omega}(g(n))$ o valgono entrambe, mentre ciò non è vero se si usa la notazione Ω invece di $\tilde{\Omega}$.

b. Descrivere i potenziali vantaggi e svantaggi dell'uso di $\tilde{\Omega}$ invece di Ω per caratterizzare il tempo di esecuzione dei programmi.

Alcuni autori definiscono O in modo leggermente diverso; si userà O' per indicare questa definizione alternativa. Si dice che $f(n) = O'(g(n))$ se e solo se $|f(n)| = O(g(n))$.

c. Cosa accade nel Teorema 2.1 alle due direzioni del "se e solo se" considerando questa nuova definizione?

Alcuni autori definiscono $\tilde{O}$ (si legge "O tilde") per indicare O con fattori logaritmici ignorati:

$$\tilde{O}(g(n)) = \{f(n)\text{: esistono costanti positive } c, k \text{ e } n_0 \text{ tali che } 0 \le f(n) \le cg(n)\lg^k(n) \text{ per ogni } n \ge n_0\}$$

d. Definire in modo simile $\tilde{\Omega}$ e $\tilde{\Theta}$. Dimostrare il corrispondente Teorema 2.1.

2-6 *Funzioni iterate*

L'operatore di iterazione "*" usato nella funzione $\lg^*$ può essere applicato a funzioni sui reali monotone crescenti. Data una funzione f per cui valga $f(n) < n$, si definisce ricorsivamente la funzione $f^{(i)}$ per interi non negativi i come

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

Per una data costante $c \in \mathbf{R}$, si definisce la funzione iterata f_c^* con

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

che non necessita di essere ben definita in tutti i casi. In altre parole, la quantità $f_c^*(n)$ è il numero di applicazioni ripetute della funzione f richieste per ridurre il suo valore fino a un valore minore o uguale a c.

Per ognuna delle seguenti funzioni $f(n)$ e costanti c, fornire un limite più stretto possibile per $f_c^*(n)$.

	$f(n)$	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 al capitolo

Knuth [121] fa risalire l'origine della notazione O ad un testo sulla teoria dei numeri di P. Bachmann del 1892. La notazione o fu introdotta da E. Landau nel 1909 nella sua trattazione della distribuzione dei numeri primi. Le notazioni Ω e Θ furono sostenute da Knuth [124] per correggere la pratica, popolare ma tecnicamente imprecisa, di usare nella letteratura la notazione O sia per limiti superiori che inferiori. Molti continuano ad usare la notazione O mentre la notazione Θ è più precisa tecnicamente. Ulteriori discussioni sulla storia e lo

sviluppo delle notazioni asintotiche si possono trovare in Knuth [121, 124] e in Brassard e Bratley [33].

Non tutti gli autori definiscono le notazioni asintotiche allo stesso modo, sebbene nelle situazioni più comuni le varie notazioni siano concordi. Alcune definizioni alternative comprendono funzioni che, anche se non sono asintoticamente non negative, hanno valori assoluti limitati in modo appropriato.

Altre proprietà di funzioni matematiche elementari si possono trovare in qualunque repertorio di matematica, come Abramowitz e Stegun [1] o Beyer [27], oppure in testi di analisi come Apostol [12] o Thomas e Finney [192]. Knuth [121] contiene abbondante materiale sulla matematica discreta usata in informatica.

Sommatorie

3

Quando un algoritmo contiene un costrutto di controllo iterativo come un ciclo **while** o **for**, il suo tempo di esecuzione può essere espresso come la somma dei tempi impiegati per ogni esecuzione del corpo del ciclo. Per esempio, si è visto nel paragrafo 1.2 che la j-esima iterazione dell'insertion sort impiega, nel caso peggiore, un tempo proporzionale a j. Sommando il tempo impiegato in ogni iterazione, si ottiene la sommatoria (o serie)

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

Valutando questa sommatoria si ottiene un limite di $\Theta(n^2)$ sul tempo di esecuzione dell'algoritmo nel caso peggiore. Questo esempio indica l'importanza di comprendere come gestire le sommatorie e fornire limiti su di esse. (Come si vedrà nel Capitolo 4, le sommatorie si ritrovano anche nell'uso di alcuni metodi per risolvere le ricorrenze.)

Il paragrafo 3.1 elenca alcune formule di base che riguardano le sommatorie. Il paragrafo 3.2 offre utili tecniche per fornire limiti sulle sommatorie. Le formule nel paragrafo 3.1 sono descritte senza dimostrazione, comunque per alcune di esse le prove sono presentate nel paragrafo 3.2 per illustrare i metodi di quel paragrafo. Molte delle altre dimostrazioni possono essere trovate in un qualunque testo di matematica.

3.1 Formule e proprietà sulle sommatorie

Data una sequenza $a_1, a_2, \ldots$ di numeri, la somma finita $a_1 + a_2 + \ldots + a_n$ può essere scritta nel seguente modo:

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

Se $n = 0$, il valore della sommatoria è 0 per definizione. Se n non è un intero, si assume che il limite superiore sia $\lfloor n \rfloor$. Analogamente se la somma comincia con $k = x$, dove x non è un intero, si assume che il valore iniziale della sommatoria sia $\lfloor x \rfloor$. (In genere, si useranno la base e il tetto esplicitamente.) Il valore di una serie finita è sempre ben definito e i suoi termini possono essere sommati in qualunque ordine.

Data una sequenza $a_1, a_2, \ldots$ di numeri, la somma infinita $a_1 + a_2 + \ldots$ può essere scritta nel seguente modo:

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

che rappresenta

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

Se il limite non esiste, la serie ***diverge***; altrimenti ***converge***. I termini di una serie convergente non sempre possono essere sommati in qualunque ordine. Si possono però risistemare i termini di una ***serie assolutamente convergente***, cioè una serie $\sum_{k=1}^{\infty} a_k$ per cui anche la serie $\sum_{k=1}^{\infty} |a_k|$ converge.

Linearità

Dato un qualunque numero reale c e due qualunque sequenze finite $a_1, a_2, \ldots, a_n$ e $b_1, b_2, \ldots, b_n$, allora

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

La proprietà di linearità si applica anche a serie convergenti infinite e può inoltre essere impiegata per manipolare sommatorie contenenti termini in notazione asintotica. Per esempio,

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

In quest'equazione, la notazione Θ sul lato sinistro è applicata alla variabile k, mentre sul lato destro è applicata ad n. Queste manipolazioni possono essere applicate a serie convergenti infinite.

Serie aritmetica

La sommatoria

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

che viene fuori dall'analisi dell'insertion sort, è una ***serie aritmetica*** ed ha come valore

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

Serie geometrica

Dato un reale $x \neq 1$, la sommatoria

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

è una ***serie geometrica*** o ***esponenziale*** ed ha come valore

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

Quando la sommatoria è infinita e $|x| < 1$, si ha la serie geometrica decrescente infinita

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

Serie armonica

Dato l'intero positivo n, l'n-esimo ***numero armonico*** è

$$\begin{aligned} H_n &= 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} \\ &= \sum_{k=1}^{n} \frac{1}{k} && (3.5) \\ &= \ln n + O(1) . \end{aligned}$$

Integrali e derivate di serie

Si possono ottenere ulteriori risultati integrando e differenziando le formule appena viste. Per esempio, derivando entrambi i lati della serie geometrica infinita (3.4) e moltiplicando per x, si ha

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

Serie telescopiche

Data una qualunque sequenza $a_0, a_1, \ldots, a_n$, vale che

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

poiché ognuno dei termini $a_1, a_2, \ldots, a_{n-1}$ è sia sommato che sottratto esattamente una volta. Si dice che la somma rientra come un cannocchiale o ***telescopicamente***. Analogamente

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

Come esempio di serie telescopica, si consideri la sommatoria

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

Poiché ogni termine può essere riscritto come

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

si ha

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

Produttorie

Il prodotto finito $a_1, a_2, \ldots, a_n$ può essere scritto come

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

Se $n = 0$, il valore del prodotto è 1 per definizione. Si può convertire una formula contenente un prodotto in una formula contenente una sommatoria usando la seguente identità:

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

Esercizi

3.1-1 Trovare una formula semplice per $\sum_{k=1}^{n}(2k-1)$.

* *3.1-2* Mostrare che $\sum_{k=1}^{n} 1/(2k-1) = \ln(\sqrt{n}) + O(1)$ manipolando le serie armoniche.

* *3.1-3* Mostrare che $\sum_{k=0}^{\infty}(k-1)/2^k = 0$.

* *3.1-4* Calcolare il valore della serie $\sum_{k=1}^{\infty}(2k+1)x^{2k}$.

3.1-5 Usare la proprietà di linearità delle serie per provare che $\sum_{k=1}^{n} O(f_k(n)) = O(\sum_{k=1}^{n} f_k(n))$.

3.1-6 Provare che $\sum_{k=1}^{\infty} \Omega(f(k)) = \Omega(\sum_{k=1}^{\infty} f(k))$.

3.1-7 Calcolare il valore del prodotto $\prod_{k=1}^{n} 2 \cdot 4^k$.

* *3.1-8* Calcolare il valore del prodotto $\prod_{k=2}^{n}(1 - 1/k^2)$.

* *3.1-9* Si dimostri che il limite della serie armonica può essere migliorato a

$$H_n = \ln n + \gamma + \frac{\epsilon}{2n} ,$$

dove $\gamma = 0.5772156649\ldots$ è conosciuta come la ***costante di Eulero-Mascheroni*** ed ϵ soddisfa $0 < \epsilon < 1$. (Si veda Knuth [121].)

3.2 Definizione di limitazioni sulle sommatorie

Vi sono molte tecniche disponibili per definire limiti sulle sommatorie che descrivono i tempi di esecuzione degli algoritmi. Di seguito saranno presentati alcuni dei metodi usati più di frequente.

Induzione matematica

Il metodo base per calcolare il valore di una serie è di usare l'induzione matematica. Per esempio, si dimostrerà che la serie aritmetica $\sum_{k=1}^{n} k$ ha come valore $\frac{1}{2}n(n+1)$. Si può facilmente verificare che è vero per $n = 1$, quindi si fa l'ipotesi induttiva che esso valga per n e si dimostra che vale per $n + 1$. Si ha

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

Per usare l'induzione matematica non è necessario tentare di determinare il valore esatto della serie. L'induzione può essere usata anche per mostrare un limite superiore. Per esempio si proverà che la serie geometrica $\sum_{k=0}^{n} 3^k$ vale $O(3^n)$. Più precisamente, si proverà che $\sum_{k=0}^{n} 3^k \le c3^n$ per qualche costante c. Per la condizione iniziale $n = 0$, vale $\sum_{k=0}^{0} 3^k = 1 \le c \cdot 1$ purché $c \ge 1$. Assumendo che il limite valga per n, si prova ora che vale per $n+1$. Si ha

$$\begin{aligned}\sum_{k=0}^{n+1} 3^k &= \sum_{k=0}^{n} 3^k + 3^{n+1} \\ &\le c3^n + 3^{n+1} \\ &= \left(\frac{1}{3} + \frac{1}{c}\right) c3^{n+1} \\ &\le c3^{n+1}\end{aligned}$$

se $(1/3 + 1/c) \le 1$ o, equivalentemente, $c \ge 3/2$. Di qui $\sum_{k=0}^{n} 3^k = O(3^n)$, come si voleva dimostrare.

Bisogna però fare attenzione a dimostrare limiti con l'induzione quando si usa la notazione asintotica. Si consideri la seguente prova fasulla che $\sum_{k=1}^{n} k = O(n)$.

Di sicuro $\sum_{k=1}^{1} k = O(1)$. Assumendo che il limite valga per n, si dimostra per $n + 1$:

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

L'errore nel procedimento è che la costante nascosta dalla O cresce con n e quindi non è una costante. Non è stato mostrato che la stessa costante vale *per ogni n*.

Limitazione dei termini

Talvolta, un buon limite superiore su una serie può essere ottenuto maggiorando ogni termine della serie, e spesso è sufficiente usare il termine più grande per limitare gli altri. Per esempio, un limite superiore immediato sulla serie aritmetica (3.1) è

$$\begin{aligned}\sum_{k=1}^{n} k &\le \sum_{k=1}^{n} n \\ &= n^2\,.\end{aligned}$$

In generale per una serie $\sum_{k=1}^{n} a_k$, se si pone $a_{\max} = \max_{1 \le k \le n} a_k$, allora vale

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

La tecnica di limitare ogni termine di una serie con il termine più grande è debole quando la serie può in realtà essere limitata da una serie geometrica. Data la serie $\sum_{k=0}^{n} a_k$ si supponga che $a_{k+1}/a_k \le r$ per ogni $k \ge 0$ dove $r < 1$ è una costante. Dato che $a_k \le a_0 r^k$ la somma può essere limitata da una serie geometrica decrescente infinita, quindi

$$\begin{aligned}\sum_{k=0}^{n} a_k &\le \sum_{k=0}^{\infty} a_0 r^k \\ &= a_0 \sum_{k=0}^{\infty} r^k \\ &= a_0 \frac{1}{1-r}\,.\end{aligned}$$

Si può applicare questo metodo per dare un limite alla sommatoria $\sum_{k=1}^{\infty} (k/3^k)$. Il primo termine è 1/3 mentre il rapporto tra due termini consecutivi è

$$\begin{aligned}\frac{(k+1)/3^{k+1}}{k/3^k} &= \frac{1}{3} \cdot \frac{k+1}{k} \\ &\le \frac{2}{3}\end{aligned}$$

per ogni $k \ge 1$. Quindi ogni termine è superiormente limitato da $(1/3)\,(2/3)^k$, e da qui si ha

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

È un errore comune, nell'applicazione di questo metodo, mostrare che il rapporto di due termini consecutivi è minore di 1 e quindi assumere che la sommatoria sia limitata da una serie geometrica. Per esempio, si potrebbe fare questo errore per la serie armonica infinita che invece diverge:

$$\begin{aligned}\sum_{k=1}^{\infty} \frac{1}{k} &= \lim_{n \to \infty} \sum_{k=1}^{n} \frac{1}{k} \\ &= \lim_{n \to \infty} \Theta(\lg n) \\ &= \infty\,.\end{aligned}$$

Il rapporto tra il $(k+1)$-esimo e il k-esimo termine di questa serie è $k/(k+1) < 1$, ma la serie non è limitata da una serie geometrica decrescente. Infatti, per limitare una serie con una serie geometrica, si deve mostrare che il rapporto è limitato da un valore strettamente inferiore a 1; cioè deve esistere una *costante* $r < 1$, tale che il rapporto fra due termini consecutivi non superi mai r. Per la serie armonica una costante r di questo tipo non esiste perché il rapporto si avvicina a 1 indefinitamente.

Spezzare le sommatorie

Per ottenere limiti su una sommatoria difficile si può esprimere la serie come la somma di due o più serie ottenute spezzando l'intervallo dell'indice e quindi limitando ognuna delle serie risultanti. Per esempio, si supponga di cercare un limite inferiore per la serie aritmetica $\sum_{k=1}^{n} k$, che come si è già visto è limitata superiormente da n^2. Si potrebbe limitare ogni termine della sommatoria con il termine più piccolo, ma poiché quel termine è 1, si avrebbe un limite inferiore per la sommatoria pari ad n, che è troppo distante dal limite superiore n^2.

Si può ottenere un risultato migliore spezzando la sommatoria. Si assuma per comodità che n sia pari. Si ha

$$\begin{aligned}\sum_{k=1}^{n} k &= \sum_{k=1}^{n/2} k + \sum_{k=n/2+1}^{n} k \\ &\geq \sum_{k=1}^{n/2} 0 + \sum_{k=n/2+1}^{n} (n/2) \\ &\geq (n/2)^2 \\ &= \Omega(n^2) ,\end{aligned}$$

che è un limite asintoticamente stretto, poiché $\sum_{k=1}^{n} k = O(n^2)$.

Spesso si può spezzare la sommatoria ottenuta dall'analisi di un algoritmo, ignorando un numero costante di termini iniziali. In genere questa tecnica si applica quando ogni termine a_k della sommatoria $\sum_{k=0}^{n} a_k$ è indipendente da n. Quindi per qualunque costante $k_0 > 0$, si può scrivere

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

poiché i termini iniziali della sommatoria sono tutti costanti e viene considerato un numero costante di essi. A questo punto si possono usare altri metodi per limitare $\sum_{k=k_0}^{n} a_k$. Per esempio, per trovare un limite asintotico superiore per

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

si osservi che il rapporto tra due termini consecutivi è

$$\begin{aligned}\frac{(k+1)^2/2^{k+1}}{k^2/2^k} &= \frac{(k+1)^2}{2k^2} \\ &\leq \frac{8}{9}\end{aligned}$$

se $k \leq 3$. Quindi la sommatoria può essere spezzata così:

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

infatti la seconda sommatoria è una serie geometrica decrescente.

La tecnica di spezzare le sommatorie può essere usata per determinare limiti asintotici in situazioni molto più difficili. Per esempio, si può ottenere un limite di $O(\lg n)$ sulla serie armonica (3.5)

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

L'idea è di suddividere l'intervallo da 1 a n in $\lfloor \lg n \rfloor$ parti e limitare superiormente con 1 il contributo di ogni parte. Per cui si ha

$$\begin{aligned}\sum_{k=1}^{n} \frac{1}{k} &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i + j} \\ &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} \sum_{j=0}^{2^i-1} \frac{1}{2^i} \\ &\leq \sum_{i=0}^{\lfloor \lg n \rfloor} 1 \\ &\leq \lg n + 1 ,\end{aligned} \tag{3.8}$$

Approssimazione con integrali

Quando una sommatoria può essere espressa come $\sum_{k=m}^{n} f(k)$, dove $f(k)$ è una funzione monotona crescente, si può approssimarla con i seguenti integrali:

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

La giustificazione di questa approssimazione è mostrata nella figura 3.1. La sommatoria rappresenta l'area dei rettangoli della figura, e l'integrale è la regione ombreggiata sotto la curva. Analogamente, quando $f(k)$ è una funzione monotona decrescente, per fornire i limiti si può usare un metodo simile:

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

L'approssimazione con l'integrale (3.10) fornisce una stima stretta per l'n-esimo numero armonico. Come limite inferiore, si ottiene

$$\begin{aligned}\sum_{k=1}^{n} \frac{1}{k} &\geq \int_{1}^{n+1} \frac{dx}{x} \\ &= \ln(n+1) ,\end{aligned} \tag{3.11}$$

Come limite superiore, si ha la disuguaglianza

$$\begin{aligned}\sum_{k=2}^{n} \frac{1}{k} &\leq \int_{1}^{n} \frac{dx}{x} \\ &= \ln n ,\end{aligned}$$

Figura 3.1 *Approssimazione di* $\sum_{k=m}^{n} f(k)$ *con integrali. Il valore dell'area di ogni rettangolo è mostrato al suo interno, e il valore totale dell'area dei rettangoli rappresenta il valore della sommatoria. L'integrale è rappresentato dall'area grigia sotto la curva.*
Confrontando le aree in (a), *si ottiene* $\int_{m-1}^{n} f(x)\,dx \le \sum_{k=m}^{n} f(k)$ *e quindi traslando di un'unità i rettangoli verso destra si ottiene* $\sum_{k=m}^{n} f(k) \le \int_{m}^{n+1} f(x)\,dx$ *in* (b).

per cui vale la limitazione

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

Esercizi

3.2-1 Mostrare che $\sum_{k=1}^{n} 1/k^2$ è limitata superiormente da una costante.

3.2-2 Trovare un limite asintotico superiore per la sommatoria

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

3.2-3 Mostrare che l'n-esimo numero armonico è $\Omega(\lg n)$ spezzando la sommatoria.

3.2-4 Approssimare $\sum_{k=1}^{n} k^3$ con un integrale.

3.2-5 Perché non si usa l'approssimazione con l'integrale (3.10) direttamente su $\sum_{k=1}^{n} 1/k$ per ottenere un limite superiore per l'n-esimo numero armonico?

Problemi

3-1 ***Limitazioni sulle sommatorie***

Dare limiti asintotici stretti sulle seguenti sommatorie. Si assuma che $r \ge 0$ e $s \ge 0$ siano costanti.

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

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

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

Note al capitolo

Knuth [121] è un riferimento eccellente per il materiale presentato in questo capitolo. Le proprietà di base delle serie possono essere trovate in qualunque buon testo di analisi matematica, come Apostol [12] o Thomas e Finney [192].

Ricorrenze

4

Quando un algoritmo contiene una chiamata ricorsiva a se stesso, come già notato nel Capitolo 1, il suo tempo di esecuzione può spesso essere descritto da una ricorrenza. Una ***ricorrenza*** è un'equazione o una disuguaglianza che descrive una funzione in termini del suo valore su input sempre più piccoli. Per esempio, nel Capitolo 1 si è mostrato come il tempo di esecuzione $T(n)$ della procedura MERGE-SORT, nel caso peggiore, possa essere descritto dalla ricorrenza

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

la cui soluzione è $T(n) = \Theta(n \lg n)$.

Questo capitolo offre tre metodi per risolvere le ricorrenze – cioè, per ottenere i limiti asintotici "Θ" o "O" della soluzione. Nel ***metodo di sostituzione*** si tenta un limite e quindi si usa l'induzione matematica per provare che il tentativo è corretto. Il ***metodo iterativo*** converte le ricorrenze in sommatorie e quindi per risolvere la ricorrenza si applicano le tecniche per limitare le sommatorie. Il ***metodo principale*** fornisce limiti sulle ricorrenze della forma

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

dove $a \geq 1$, $b > 1$ e $f(n)$ è una funzione; il metodo richiede di ricordare tre casi, ma consente di determinare facilmente i limiti asintotici di molte semplici ricorrenze.

Aspetti tecnici

Nella pratica, quando si descrivono e risolvono le ricorrenze, si trascurano certi dettagli tecnici. Per esempio, un dettaglio che viene spesso mascherato è l'ipotesi che le funzioni abbiano argomenti interi. Di solito, il tempo di esecuzione $T(n)$ di un algoritmo è definito solo su n intero, infatti per la maggior parte degli algoritmi, la dimensione dell'input è sempre un intero. Per esempio, la ricorrenza che descrive il tempo di esecuzione del MERGE-SORT nel caso peggiore è in realtà

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

Le condizioni al contorno rappresentano un'altra classe di dettagli tipicamente ignorati. Poiché il tempo di esecuzione di un algoritmo è costante su un input di dimensione costante, le ricorrenze che descrivono tale tempo hanno in genere $T(n) = \Theta(1)$ per n sufficientemente

piccolo. Di conseguenza, per comodità, in generale si ometteranno le descrizioni delle condizioni al contorno delle ricorrenze e si assumerà che $T(n)$ è costante per n piccolo. Per esempio, normalmente si descrive la ricorrenza (4.1) come

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

senza dare esplicitamente i valori per n piccolo. La ragione è che, sebbene cambiando il valore di $T(1)$ cambi la soluzione della ricorrenza, tipicamente la soluzione non cambia per più di un fattore costante, così che l'ordine di grandezza rimane invariato.

Quando si descrivono e si risolvono le ricorrenze, si omettono spesso la base, il tetto e le condizioni al contorno. Si procederà per gradi, prima senza considerare questi dettagli, e poi determinando se hanno o no importanza. Di solito non ne hanno, tuttavia è bene sapere quando ne hanno. L'esperienza aiuta, così come alcuni teoremi che stabiliscono che questi dettagli non influiscono sui limiti asintotici di molte ricorrenze incontrate nell'analisi degli algoritmi (si veda il Teorema 4.1 e il Problema 4-5). In questo capitolo, comunque, si farà riferimento ad alcuni di questi dettagli per mostrare le sottigliezze dei metodi per risolvere le ricorrenze.

4.1 Il metodo di sostituzione

Il metodo di sostituzione per risolvere le ricorrenze richiede di tentare uno schema di soluzione, quindi usare l'induzione matematica per trovare le costanti e mostrare che la soluzione funziona. Il nome viene dalla *sostituzione* nella funzione della soluzione tentata quando l'ipotesi induttiva è applicata a valori più piccoli. Questo metodo è potente, ma può essere applicato solo quando sia facile azzeccare lo schema della soluzione.

Il metodo di sostituzione può essere usato per stabilire i limiti inferiore o superiore di una ricorrenza. Per esempio, determiniamo un limite superiore per la ricorrenza

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

che è simile alle ricorrenze (4.2) e (4.3). Si tenta con la soluzione $T(n) = O(n \lg n)$. Il metodo consiste nel provare che $T(n) \le cn \lg n$ dove c è una costante positiva scelta in modo appropriato. Assumendo che questo limite valga per $\lfloor n/2 \rfloor$, cioè che $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$, e sostituendo nella ricorrenza si ha

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

dove l'ultimo passo vale per $c \ge 1$.

L'induzione matematica richiede ora di mostrare che la soluzione vale per le condizioni al contorno, cioè si deve mostrare che si può scegliere la costante c sufficientemente grande in modo che $T(n) \le cn \lg n$ valga anche per le condizioni al contorno. Questo requisito può talvolta creare dei problemi. Si assuma che $T(1) = 1$ sia l'unica condizione al contorno della ricorrenza. Allora, sfortunatamente, non si può scegliere c sufficientemente grande, perché $T(1) > c\,1 \lg 1 = 0$.

Questa difficoltà nel provare un'ipotesi induttiva per una specifica condizione al contorno può essere facilmente superata. Si può trarre vantaggio dal fatto che la notazione asintotica richiede soltanto di provare che $T(n) \le cn \lg n$ per $n \ge n_0$, dove n_0 è una costante. L'idea è di trascurare nella prova induttiva la condizione al contorno $T(1) = 1$ e di considerare nella prova $n = 2$ e $n = 3$ come parte delle condizioni al contorno. Si possono considerare $T(2)$ e $T(3)$ come condizioni al contorno perché per $n > 3$, la ricorrenza non dipende direttamente da $T(1)$. Dalla ricorrenza si trova che $T(2) = 4$ e $T(3) = 5$. La prova induttiva che $T(n) \le cn \lg n$ per qualche costante $c \ge 1$ può essere completata scegliendo c sufficientemente grande perché $T(2) \le c2\lg 2$ e $T(3) \le c3\lg 3$. Come si può verificare è sufficiente qualunque scelta di $c \ge 2$. Per molte delle ricorrenze che verranno esaminate è semplice estendere le condizioni al contorno perché l'ipotesi induttiva funzioni per n piccolo.

Come scegliere la soluzione candidata

Sfortunatamente, non esiste una regola generale per azzeccare la soluzione corretta per ogni ricorrenza. Fortunatamente, però, vi sono alcune euristiche che possono aiutare il lettore a diventare un buon risolutore.

Se una ricorrenza è simile a quelle viste prima, allora tentare una soluzione simile è ragionevole. Per esempio si consideri la ricorrenza

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

che sembra difficile a causa del "17" aggiunto sul lato destro all'argomento di T. Intuitivamente questo termine addizionale non può influenzare in modo sostanziale la soluzione della ricorrenza. Quando n è grande la differenza tra $T(\lfloor n/2 \rfloor)$ e $T(\lfloor n/2 \rfloor + 17)$ non lo è altrettanto: entrambe dividono n all'incirca a metà. Di conseguenza, stentano con $T(n) = O(n \lg n)$, che si può dimostrare corretta usando il metodo di sostituzione (si veda l'Esercizio 4.1-5).

Un altro modo per azzeccare la soluzione è di provare limiti laschi della ricorrenza e quindi ridurre il grado di incertezza. Per esempio, si potrebbe cominciare con un limite inferiore di $T(n) = \Omega(n)$ per la ricorrenza (4.4), poiché il termine n è nella ricorrenza, e provare un limite superiore iniziale di $T(n) = O(n^2)$. Quindi si può gradualmente abbassare il limite superiore e alzare il limite inferiore finché convergano alla soluzione corretta e asintoticamente stretta $T(n) = \Theta(n \lg n)$.

Sottigliezze

A volte si può azzeccare correttamente un limite asintotico della soluzione di una ricorrenza, ma, in qualche modo, può sembrare che il calcolo matematico non torni nell'induzione. Di solito, il problema è che l'ipotesi induttiva non sia abbastanza forte da dimostrare limiti specifici. Quando ci si imbatte in una difficoltà di questo tipo, correggere la soluzione sottraendo un termine di ordine più basso, spesso consente di ottenere la prova matematica.

Si consideri la ricorrenza

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

Si tenta con la soluzione $O(n)$, e si cerca di mostrare che $T(n) \le cn$ dove c è una costante scelta appropriatamente. Sostituendo nella ricorrenza, si ha

$$\begin{aligned} T(n) &\le c\lfloor n/2\rfloor + c\lceil n/2\rceil + 1 \\ &= cn + 1 , \end{aligned}$$

che non implica $T(n) \le cn$ per nessuna scelta di c. Sarebbe interessante cercare una soluzione più grande, che può funzionare, come $T(n) = O(n^2)$, anche se in realtà l'ipotesi che la soluzione sia $T(n) = O(n)$ è corretta. Per dimostrare questa, però, bisogna fare un'ipotesi induttiva più forte.

Intuitivamente l'ipotesi è esatta: essa non torna solo per la costante 1, un termine di ordine più basso. Ciò nonostante l'induzione matematica non funziona se non si prova la forma esatta dell'ipotesi induttiva. Si supera la difficoltà *sottraendo* un termine di ordine più basso dal precedente tentativo. Il nuovo tentativo è $T(n) \le cn - b$ dove $b \ge 0$ è una costante. Si ha

$$\begin{aligned} T(n) &\le (c\lfloor n/2\rfloor - b) + (c\lceil n/2\rceil - b) + 1 \\ &= cn - 2b + 1 \\ &\le cn - b , \end{aligned}$$

per $b \ge 1$. Come prima, la costante c deve essere scelta sufficientemente grande da gestire le condizioni al contorno.

Molte persone trovano controintuitiva l'idea di *sottrarre* un termine di ordine inferiore. Dopotutto se il calcolo matematico non torna, la soluzione non dovrebbe essere *aumentata*? La chiave di comprensione di questo passaggio sta nel ricordare che si usa l'induzione: si può dimostrare qualcosa di più forte per un valore dato assumendo qualcosa di più forte per valori più piccoli.

Evitare le trappole

È facile sbagliare l'uso della notazione asintotica. Per esempio, nella ricorrenza (4.4) si può provare, erroneamente, che $T(n) = O(n)$ tentando con $T(n) \le cn$ e quindi dedurre che

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

poiché c è una costante. L'errore è che non è stata dimostrata la forma esatta dell'ipotesi induttiva e cioè che $T(n) \le cn$.

Sostituzione di variabili

Talvolta, una piccola manipolazione algebrica può rendere una ricorrenza sconosciuta simile a una già vista. Per esempio si consideri la ricorrenza

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

che sembra difficile. Tuttavia si può semplificare questa ricorrenza con una sostituzione di variabile. Per comodità, si trascurerà l'arrotondamento a interi dei valori come $\sqrt{n}$. Sostituendo $m = \lg n$ si ottiene

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

Si può ridenominare $S(m) = T(2^m)$ per produrre la nuova ricorrenza

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

che è molto simile alla ricorrenza (4.4) ed ha la stessa soluzione: $S(m) = O(m \lg m)$. Risostituendo le variabili ridenominate, si ha $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg\lg n)$.

Esercizi

4.1-1 Mostrare che la soluzione di $T(n) = T(\lceil n/2\rceil) + 1$ è $O(\lg n)$.

4.1-2 Mostrare che la soluzione di $T(n) = 2T(\lfloor n/2\rfloor) + n$ è $\Omega(n \lg n)$. Concludere che la soluzione è $\Theta(n \lg n)$.

4.1-3 Mostrare che facendo una diversa ipotesi induttiva, si può superare la difficoltà della condizione al contorno $T(1) = 1$ per la ricorrenza (4.4) senza la correzione sulle condizioni al contorno per la prova induttiva.

4.1-4 Mostrare che $\Theta(n \lg n)$ è la soluzione della ricorrenza "esatta" (4.2) per il merge-sort.

4.1-5 Mostrare che la soluzione di $T(n) = 2T(\lfloor n/2\rfloor + 17) + n$ è $O(n \lg n)$.

4.1-6 Risolvere la ricorrenza $T(n) = 2T(\sqrt{n}) + 1$ facendo una sostituzione di variabili. Non preoccuparsi che i valori siano interi.

4.2 Il metodo iterativo

Il metodo iterativo non richiede di tentare una soluzione, ma può richiedere molta più algebra del metodo di sostituzione. L'idea è di sviluppare (iterare) la ricorrenza ed esprimerla come somma di termini dipendenti solo da n e dalle condizioni iniziali. Le tecniche per valutare le sommatorie possono essere usate per fornire limiti alla soluzione.

Per esempio, si consideri la ricorrenza

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

Si itera come segue:

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

dove $\lfloor\lfloor n/4\rfloor/4\rfloor = \lfloor n/16\rfloor$ e $\lfloor\lfloor n/16\rfloor/4\rfloor = \lfloor n/64\rfloor$ vengono dall'identità (2.4).

Quanto bisogna sviluppare la ricorrenza per raggiungere una condizione al contorno? Il termine i-esimo nelle serie è $3^i\lfloor n/4^i\rfloor$. L'iterazione raggiunge il valore 1 quando $\lfloor n/4^i\rfloor = 1$ o, equivalentemente, i raggiunge $\log_4 n$. Proseguendo nell'iterazione fino a questo punto e usando la limitazione $\lfloor n/4^i\rfloor \le n/4^i$ si scopre che la sommatoria contiene una serie geometrica decrescente:

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

In questo caso, è stata usata l'identità (2.9) per concludere che $3^{\log_4 n} = n^{\log_4 3}$, e il fatto che $\log_4 3 < 1$ per concludere che $\Theta(n^{\log_4 3}) = o(n)$.

Il metodo iterativo di solito comporta l'uso di molta algebra, e fare tutti i calcoli in modo esatto può essere complesso. Bisogna concentrarsi su due parametri: il numero di volte che la ricorrenza deve essere iterata per raggiungere le condizioni al contorno, e la somma dei termini originati da ogni livello del processo di iterazione. Talvolta, nel processo di iterazione di una ricorrenza, si può azzeccare la soluzione senza sviluppare tutti i calcoli matematici. in tal caso, il metodo iterativo può essere abbandonato per scegliere il metodo di sostituzione, che di solito richiede un uso inferiore di algebra.

Quando una ricorrenza contiene funzioni base e tetto, il calcolo matematico può diventare particolarmente complicato. Spesso, può aiutare assumere che la ricorrenza sia definita solo sulla potenza esatta di un numero. Nell'esempio, se si fosse assunto che $n = 4^k$ per qualche intero k, la funzione *base* avrebbe potuto essere comodamente omessa. Sfortunatamente, mostrare il limite $T(n) = O(n)$ esclusivamente per potenze esatte di 4 è tecnicamente un uso improprio della notazione O. La definizione di notazione asintotica richiede che i limiti siano provati per *qualunque* intero sufficientemente grande, non solo per le potenze di 4. Si vedrà nel Paragrafo 4.3 che per una ampia classe di ricorrenze, questi aspetti tecnici possono essere superati. Il Problema 4-5 illustra alcune condizioni valendo le quali l'analisi sulle potenze esatte di un intero può essere estesa a tutti gli interi.

Alberi di ricorsione

Un *albero di ricorsione* è un modo comodo di visualizzare ciò che accade quando si itera una ricorrenza, e che può aiutare a organizzare la gestione algebrica necessaria a risolvere la ricorrenza. È utile in special modo quando la ricorrenza descrive un algoritmo divide-et impera. La figura 4.1 mostra la derivazione dell'albero di ricorsione per

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

Per comodità, si assuma che n sia una potenza esatta di 2. La parte (a) della figura mostra $T(n)$, che nella parte (b) è stata espansa in un albero equivalente che rappresenta la ricorrenza. Il termine n^2 è la radice (l'etichetta al primo livello di ricorsione), e i due sottoalberi sono le due ricorrenze $T(n/2)$. La parte (c) mostra lo stesso processo portato avanti di un passo espandendo $T(n/2)$. L'etichetta dei due nodi al secondo livello di ricorsione è $(n/2)^2$. Si espande ancora ogni nodo dell'albero espandendo le parti che lo costituiscono descritte dalla ricorrenza, finché non è raggiunta una condizione al contorno. La parte (d) mostra l'albero risultante.

Per valutare la ricorrenza si sommano i valori su ogni livello dell'albero. Il primo livello ha come valore totale n^2, il secondo livello ha come valore $(n/2)^2 + (n/2)^2 = n^2/2$, il terzo livello ha come valore $(n/4)^2 + (n/4)^2 + (n/4)^2 + (n/4)^2 = n^2/4$ e così via. Poiché il valore decresce

Figura 4.1 *Costruzione di un albero di ricorsione per la ricorrenza* $T(n) = 2T(n/2) + n^2$. *La parte* (a) *mostra* $T(n)$ *che è progressivamente espanso in* (b) - (d) *per formare l'albero di ricorsione. L'albero espanso completamente nella parte* (d) *ha altezza* lg*n* (*ha* lg*n* + 1 *livelli*).

geometricamente, il valore totale differisce, al più, di un fattore costante dal termine più grande (il primo), quindi la soluzione è $\Theta(n^2)$.

La figura 4.2 mostra un altro esempio più intricato, l'albero di ricorsione per la ricorrenza

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

(Anche in questo caso le funzioni base e tetto sono omesse per semplicità.) Quando si sommano i valori nei livelli dell'albero di ricorsione, si ha un valore n per ogni livello. Il cammino più lungo dalla radice alla foglia è $n \to (2/3)n \to (2/3)^2 n \to \ldots \to 1$. Poiché $(2/3)^k n = 1$ per $k = \log_{3/2} n$, l'altezza dell'albero è $\log_{3/2} n$. Quindi la soluzione della ricorrenza è al più $n \log_{3/2} n = O(n \lg n)$.

Esercizi

4.2-1 Determinare tramite l'iterazione un buon limite asintotico superiore per la ricorrenza $T(n) = 3T(\lfloor n/2 \rfloor) + n$.

4.2-2 Dedurre che la soluzione della ricorrenza $T(n) = T(n/3) + T(2n/3) + n$ è $\Omega(n \lg n)$ utilizzando un albero di ricorsione.

Figura 4.2 *Un albero di ricorsione per la ricorrenza $T(n) = T(n/3) + T(2n/3) + n$.*

4.2-3 Tracciare l'albero di ricorsione per $T(n) = 4T(\lfloor n/2 \rfloor) + n$ e fornire limiti asintotici stretti per la soluzione.

4.2-4 Usare l'iterazione per risolvere la ricorrenza $T(n) = T(n - a) + T(a) + n$, dove $a \geq 1$ è una costante.

4.2-5 Usare un albero di ricorsione per risolvere la ricorrenza $T(n) = T(\alpha n) + T((1 - \alpha)n) + n$, dove α è una costante nell'intervallo $0 < \alpha < 1$.

4.3 Il metodo principale

Questo metodo fornisce un approccio tipo "ricettario" per risolvere le ricorrenze della forma

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

dove $a \geq 1$ e $b > 1$ sono costanti e $f(n)$ è una funzione asintoticamente positiva. Il metodo richiede di ricordare a memoria tre casi, ma la soluzione di molte ricorrenze può essere determinata molto facilmente, spesso senza carta e penna.

La ricorrenza (4.5) descrive il tempo di esecuzione di un algoritmo che divide un problema di dimensione n in a sottoproblemi, ognuno di dimensione n/b, dove a e b sono costanti positive. Gli a sottoproblemi sono risolti ricorsivamente, ognuno con tempo $T(n/b)$. Il costo di dividere il problema e combinare i risultati dei sottoproblemi è descritto dalla funzione $f(n)$. (Cioè, usando la notazione del Paragrafo 1.3.2, $f(n) = D(n) + C(n)$.) Per esempio la ricorrenza originata dalla procedura MERGE-SORT ha $a = 2$, $b = 2$ e $f(n) = \Theta(n)$.

Nel rispetto della correttezza tecnica, la ricorrenza in effetti non è ben definita perché n/b potrebbe non essere un intero. Tuttavia, la sostituzione di ognuno degli a termini $T(n/b)$ con $T(\lfloor n/b \rfloor)$ o con $T(\lceil n/b \rceil)$ non influisce sul comportamento asintotico della ricorrenza. (Sarà mostrato nel prossimo paragrafo.) In condizioni normali è comodo, dunque, omettere le funzioni base e tetto quando si scrivono ricorrenze divide-et-impera di questa forma.

Il teorema principale

Il metodo principale dipende dal seguente teorema.

Teorema 4.1 (Teorema principale)

Siano $a \geq 1$ e $b > 1$ costanti e $f(n)$ una funzione, e $T(n)$ sia definito sugli interi non negativi dalla ricorrenza:

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

dove n/b rappresenta $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$. Allora $T(n)$ può essere asintoticamente limitato come segue.

1. Se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche costante $\varepsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per qualche costante $\varepsilon > 0$, e se $af(n/b) < cf(n)$ per qualche costante $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$.

Prima di applicare il teorema principale a qualche esempio, cercheremo di capire che cosa dice. In ognuno dei tre casi si confronta la funzione $f(n)$ con la funzione $n^{\log_b a}$. Intuitivamente, la soluzione della ricorrenza è determinata dalla più grande tra le due funzioni. Se, come nel caso 1, la funzione $n^{\log_b a}$ è la più grande, allora la soluzione è $T(n) = \Theta(n^{\log_b a})$. Se, come nel caso 3, la funzione $f(n)$ è la più grande, allora la soluzione è $T(n) = \Theta(f(n))$. Se, come nel caso 2, le due funzioni hanno la stessa grandezza, si moltiplica per un fattore logaritmico, e la soluzione è $T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(f(n) \lg n)$.

Oltre questo concetto intuitivo, bisogna comprendere alcuni aspetti tecnici. Nel primo caso, non solo $f(n)$ deve essere più piccola di $n^{\log_b a}$, ma deve esserlo in modo *polinomiale*. Cioè, $f(n)$ deve essere asintoticamente più piccola di $n^{\log_b a}$ per un fattore di n^{ε} per qualche costante $\varepsilon > 0$. Nel terzo caso, non solo $f(n)$ deve essere più grande di $n^{\log_b a}$, ma deve esserlo in modo polinomiale e inoltre deve soddisfare la condizione di "regolarità" per cui $af(n/b) \leq cf(n)$. Questa condizione è soddisfatta dalla maggior parte delle funzioni polinomialmente limitate che si incontreranno.

È importante sottolineare che i tre casi non comprendono tutte le possibilità per $f(n)$; tra i casi 1 e 2 vi è un intervallo in cui $f(n)$ è più piccola di $n^{\log_b a}$ ma non lo è in modo polinomiale. Analogamente, vi è un intervallo tra i casi 2 e 3 in cui $f(n)$ è più grande di $n^{\log_b a}$ ma non in modo polinomiale. Se la funzione $f(n)$ ricade in uno di questi intervalli, o se la condizione di regolarità del caso 3 non è soddisfatta, il metodo principale non può essere usato per risolvere la ricorrenza.

Uso del metodo principale

Per usare il metodo principale, si determina semplicemente quale caso (se ve ne è uno) del teorema principale si applica e si scrive la risposta.

· Come primo esempio si consideri

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

Per questa ricorrenza, si ha $a = 9$, $b = 3$ e $f(n) = n$, e quindi $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Poiché $f(n) = O(n^{\log_3 9 - \varepsilon})$ dove $\varepsilon = 1$, si può applicare il caso 1 del teorema principale e concludere che la soluzione è $T(n) = \Theta(n^2)$.

Si consideri ora

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

in cui $a = 1$, $b = 3/2$, $f(n) = 1$ e $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Poiché $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, si applica il caso 2 e quindi la soluzione della ricorrenza è $T(n) = \Theta(\lg n)$.

Per la ricorrenza

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

si ha $a = 3$, $b = 4$, $f(n) = n \lg n$ e $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Poiché $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ dove $\varepsilon \approx 0.2$, si applica il caso 3 se si riesce a dimostrare che la condizione di regolarità su $f(n)$ è soddisfatta. Per n sufficientemente grande, $af(n/b) = 3(n/4)\lg(n/4) \le (3/4)n \lg n = cf(n)$ con $c = 3/4$. Di conseguenza, dal caso 3, la soluzione della ricorrenza è $T(n) = \Theta(n \lg n)$.

Il metodo principale non è applicabile alla ricorrenza

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

anche se presenta una forma appropriata: $a = 2$, $b = 2$, $f(n) = n \lg n$ e $n^{\log_b a} = n$. Poiché $f(n) = n \lg n$ è asintoticamente più grande di $n^{\log_b a} = n$, sembra che si possa applicare il caso 3, ma non è possibile perché non lo è in modo *polinomiale*. Il rapporto $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ è asintoticamente minore di n^ε per qualunque costante ε positiva. Conseguentemente, la ricorrenza ricade nell'intervallo tra il caso 2 e il caso 3. (Si veda l'Esercizio 4.4-2 per una soluzione).

Esercizi

4.3-1 Usare il metodo principale per dare un limite asintotico stretto alle seguenti ricorrenze:

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 Il tempo di esecuzione di un algoritmo A è descritto dalla ricorrenza $T(n) = 7T(n/2) + n^2$.

Un altro algoritmo A' ha un tempo di esecuzione di $T'(n) = aT'(n/4) + n^2$. Qual è il più grande valore intero di a tale che A' sia asintoticamente più veloce di A?

4.3-3 Usare il metodo principale per mostrare che la soluzione della ricorrenza $T(n) = T(n/2) + \Theta(1)$ della ricerca binaria (si veda l'Esercizio 1.3-5) è $T(n) = \Theta(\lg n)$.

* *4.3-4* Considerata la condizione di regolarità $af(n/b) \le cf(n)$ per qualche costante $c < 1$, che è prevista nel caso 3 del teorema principale, si dia un esempio di una semplice funzione $f(n)$ che soddisfi tutte le condizioni del caso 3 del teorema principale eccetto la condizione di regolarità.

* 4.4 Dimostrazione del teorema principale

Questo paragrafo contiene la dimostrazione del teorema principale (Teorema 4.1) per i lettori più interessati. Non è necessario comprendere la dimostrazione per applicare il teorema.

La dimostrazione consiste di due parti. La prima analizza la ricorrenza "principale" (4.5), assumendo per semplicità che $T(n)$ sia definita solo su potenze esatte di $b > 1$, cioè per $n = 1$, b, b^2, ...; questa parte fornisce tutti i concetti intuitivi necessari a comprendere perché il teorema è vero. La seconda parte mostra come l'analisi possa essere estesa a tutti gli interi n positivi e consiste semplicemente nell'applicazione di tecnica matematica per gestire base e tetto.

In questo paragrafo, talvolta la notazione asintotica sarà usata in modo leggermente improprio per descrivere il comportamento di funzioni che sono definite solo su potenze esatte di b. Si ricorda che le definizioni di notazione asintotica richiedono che i limiti siano dimostrati per tutti i numeri sufficientemente grandi, e non solo per le potenze di b. Dato che si potrebbe definire una nuova notazione asintotica che si applica all'insieme $\{b^i: i = 0, 1, \ldots\}$, invece che a interi non negativi, questo uso improprio non è grave.

Tuttavia, bisogna sempre fare attenzione quando si usa la notazione asintotica su un dominio limitato per non trarre conclusioni scorrette. Per esempio, la dimostrazione che $T(n) = O(n)$ quando n è una potenza esatta di 2 non garantisce che $T(n) = O(n)$, infatti la funzione potrebbe essere definita così:

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

nel qual caso il miglior limite superiore che possa essere dimostrato è $T(n) = O(n^2)$. A causa di drastiche conseguenze di questo tipo, non sarà mai usata la notazione asintotica su domini limitati senza che sia assolutamente chiaro dal contesto che lo si sta facendo.

4.4.1 Dimostrazione per potenze esatte

La prima parte della dimostrazione del teorema principale analizza la ricorrenza principale (4.5)

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

sotto l'ipotesi che n sia una potenza esatta di $b > 1$, dove b non deve necessariamente essere un intero. L'analisi si articola in tre lemmi; il primo riduce il problema di risolvere la ricorrenza principale al problema di valutare un'espressione che contiene una sommatoria; il secondo determina i limiti su questa sommatoria; il terzo lemma mette insieme i primi due per dimostrare una versione del teorema principale nel caso in cui n sia una potenza esatta di b.

Lemma 4.2

Siano $a \ge 1$ e $b > 1$ due costanti e $f(n)$ una funzione non negativa definita su potenze esatte di b e si definisca $T(n)$ su potenze esatte di b con la ricorrenza

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

dove i è un intero positivo: allora

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

Dimostrazione. Iterando la ricorrenza si ottiene

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

Poiché $a^{\log_b n} = n^{\log_b a}$, usando la condizione al contorno $T(1) = \Theta(1)$, l'ultimo termine del l'espressione diventa

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

I restanti termini possono essere espressi con la sommatoria

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

da cui deriva

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

che completa la dimostrazione. ■

L'albero di ricorsione

Prima di proseguire, cercheremo di giustificare intuitivamente la dimostrazione precedente usando un albero di ricorsione. La Fig. 4.3 mostra l'albero corrispondente all'iterazione della ricorrenza nel Lemma 4.2. La radice dell'albero ha costo $f(n)$, ed ha a figli, ognuno con costo $f(n/b)$. (Conviene pensare ad a come ad un numero intero, specialmente quando si utilizza l'albero di ricorsione, anche se da un punto di vista matematico ciò non è richiesto). Ognuno di questi figli ha a sua volta a figli con costo $f(n/b^2)$, quindi ci sono a^2 nodi a distanza 2 dalla radice. In generale, vi sono a^j nodi a distanza j dalla radice, ed ognuno ha costo $f(n/b^j)$. Il costo di ogni foglia è $T(1) = \Theta(1)$, e ogni foglia è distante $\log_b n$ dalla radice, poiché $n/b^{\log_b n} = 1$. Nell'albero vi sono $a^{\log_b n} = n^{\log_b a}$ foglie.

Si può ottenere l'equazione (4.6) sommando i costi di ogni livello dell'albero, come mostrato nella figura. Il costo dei nodi interni del livello j è $a^j f(n/b^j)$, e quindi il totale dei costi su tutti i livelli di nodi interni è

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

Figura 4.3 *L'albero di ricorsione generato per la ricorrenza $T(n) = aT(n/b) + f(n)$. L'albero è a-ario completo con $n^{\log_b a}$ foglie e altezza $\log_b n$. Il costo di ogni livello è mostrato sulla destra, e la somma dei costi è data nell'equazione* (4.6).

In un algoritmo divide-et-impera, questa somma rappresenta il costo della suddivisione del problema in sottoproblemi e della loro ricombinazione. Il costo di tutte le foglie, che è il costo di tutti gli $n^{\log_b a}$ sottoproblemi di dimensione 1, è $\Theta(n^{\log_b a})$.

In termini di albero di ricorsione, i tre casi del teorema principale corrispondono ai casi in cui il costo totale dell'albero è (1) dominato dal costo delle foglie, (2) parimenti distribuito su tutti i livelli dell'albero, (3) dominato dal costo della radice.

La sommatoria nell'equazione (4.6) descrive il costo dei passi di divisione e ricombinazione di un algoritmo divide-et-impera. Il prossimo lemma fornisce i limiti asintotici sulla crescita della sommatoria.

Lemma 4.3

Siano $a \geq 1$ e $b > 1$ due costanti e $f(n)$ una funzione non negativa definita su potenze esatte di b. Una funzione $g(n)$, definita su potenze esatte di b nel modo seguente,

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

può essere limitata asintoticamente per potenze esatte di b come segue.

1. Se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche costante $\varepsilon > 0$, allora $g(n) = O(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$, allora $g(n) = \Theta(n^{\log_b a} \lg n)$
3. Se $af(n/b) \le cf(n)$ per qualche costante $c < 1$ e per ogni $n \ge b$, allora $g(n) = \Theta(f(n))$.

Dimostrazione. Per il caso 1, si ha $f(n) = O(n^{\log_b a - \varepsilon})$, che implica $f(n/b^j) = O((n/b^j)^{\log_b a - \varepsilon})$. Sostituendo nell'equazione (4.7) si ottiene

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

Esplicitiamo la sommatoria interna alla notazione O portando fuori un fattore, semplificando e calcolando la serie geometrica crescente ottenuta:

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

Poiché b ed ε sono costanti, l'ultima espressione si riduce a $n^{\log_b a - \varepsilon} O(n^\varepsilon) = O(n^{\log_b a})$. Sostituendo questa espressione alla sommatoria nell'equazione (4.8) si ottiene

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

e il caso 1 è dimostrato.

Sotto l'ipotesi che $f(n) = \Theta(n^{\log_b a})$ per il caso 2, si ottiene che $f(n/b^j) = \Theta((n/b^j)^{\log_b a})$. Sostituendo nell'equazione (4.7) si ha

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

Si limita la sommatoria dentro il Θ come nel caso 1, ma questa volta non si ottiene una serie geometrica, anzi si scopre che ogni termine della sommatoria è lo stesso:

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

Sostituendo questa espressione alla sommatoria nell'equazione (4.9) si ottiene

$$\begin{aligned}
g(n) &= \Theta(n^{\log_b a} \log_b n) \\
&= \Theta(n^{\log_b a} \lg n) ,
\end{aligned}$$

e il caso 2 è dimostrato.

Il caso 3 viene dimostrato analogamente. Poiché $f(n)$ compare nella definizione (4.7) di $g(n)$ e tutti i termini di $g(n)$ sono non negativi, si può concludere che $g(n) = \Omega(f(n))$ per potenze esatte di b. Sotto l'ipotesi che $af(n/b) \le cf(n)$ per qualche costante $c < 1$ e per ogni $n \ge b$, si ha $a^j f(n/b^j) \le c^j f(n)$.

Sostituendo nell'equazione (4.7) e semplificando si ottiene una serie geometrica, ma diversamente dalla serie del caso 1, questa ha termini decrescenti:

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

poiché c è una costante. Di qui si può concludere che $g(n) = \Theta(f(n))$ per potenze esatte di b. Questa dimostrazione del caso 3 completa la dimostrazione del lemma. ■

Possiamo ora dimostrare una versione del teorema principale per il caso in cui n è una potenza esatta di b.

Lemma 4.4

Siano $a \ge 1$ e $b > 1$ due costanti e $f(n)$ una funzione non negativa definita su potenze esatte di b e si definisca $T(n)$ su potenze esatte di b con la ricorrenza

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

dove i è un intero positivo. Allora $T(n)$ può essere limitato asintoticamente per potenze esatte di b come segue.

1. Se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche costante $\varepsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$.
2. Se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per qualche costante $\varepsilon > 0$, e se $af(n/b) \le cf(n)$ per qualche costante $c < 1$ e per ogni n sufficientemente grande, allora $T(n) = \Theta(f(n))$.

Dimostrazione. Useremo i limiti stabiliti dal lemma 4.3 per valutare la sommatoria (4.6) del lemma 4.2. Per il caso 1, si ha

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

e per il caso 2

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

Per il caso 3, la condizione $af(n/b) \le cf(n)$ implica che $f(n) = \Omega(n^{\log_b a + \varepsilon})$ (Si veda l'Esercizio 4.4-3). Di conseguenza

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

■

4.4.2 Base e tetto

Per completare la dimostrazione del teorema principale, si deve estendere l'analisi alla situazione in cui nella ricorrenza principale sono usati base e tetto, in modo che la ricorrenza sia definita su tutti gli interi e non solo su potenze esatte di b. Un limite inferiore per

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

e un limite superiore per

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

si ottengono in modo semplice: infatti, per ottenere il risultato desiderato, si può usare il limite $\lceil n/b \rceil \ge n/b$ nel primo caso e il limite $\lfloor n/b \rfloor \le n/b$ nel secondo. Limitare inferiormente la ricorrenza (4.11) richiede tecniche analoghe a quelle per la limitazione superiore della ricorrenza (4.10), così sarà presentato solo quest'ultimo limite.

Si itera la ricorrenza (4.10), come già fatto per il lemma 4.2; si ottiene una sequenza di applicazioni ricorsive sugli argomenti seguenti:

$$\begin{aligned} & n , \\ & \lceil n/b \rceil , \\ & \lceil \lceil n/b \rceil / b \rceil , \\ & \lceil \lceil \lceil n/b \rceil / b \rceil / b \rceil , \\ & \quad \vdots \end{aligned}$$

Si denoti l'i-esimo elemento della sequenza con n_i, dove

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

Il primo obiettivo è di determinare il numero di iterazioni k tale che n_k sia una costante. Usando la disuguaglianza $\lceil x \rceil \le x + 1$, si ottiene

$$\begin{aligned} n_0 &\le n , \\ n_1 &\le \frac{n}{b} + 1 , \\ n_2 &\le \frac{n}{b^2} + \frac{1}{b} + 1 , \\ n_3 &\le \frac{n}{b^3} + \frac{1}{b^2} + \frac{1}{b} + 1 , \\ &\vdots \end{aligned}$$

In generale,

$$\begin{aligned} n_i &\le \frac{n}{b^i} + \sum_{j=0}^{i-1} \frac{1}{b^j} \\ &\le \frac{n}{b^i} + \frac{b}{b-1} , \end{aligned}$$

e quindi, quando $i = \lfloor \log_b n \rfloor$, si ottiene $n_i \le b + b/(b-1) = O(1)$.

Si può ora iterare la ricorrenza (4.10), ottenendo

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

che è molto simile all'equazione (4.6), dove però n può essere un qualunque intero e non soltanto una potenza esatta di b.

Si può ora valutare la sommatoria

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

dalla (4.13) in modo analogo alla dimostrazione del lemma 4.3.

Iniziando con il caso 3, se $af(\lceil n/b \rceil) \le cf(n)$ per $n > b + b/(b-1)$, dove $c < 1$ è una costante, allora segue che $a^j f(n_j) \le c^j f(n)$. Di conseguenza la sommatoria nell'equazione (4.14) può essere valutata come nel lemma 4.3. Per il caso 2 si ha $f(n) = \Theta(n^{\log_b a})$; se si riesce a dimostrare che $f(n_j) = O(n^{\log_b a}/a^j) = O((n/b^j)^{\log_b a})$ allora la prova del caso 2 del lemma 4.3 andrà bene. Si

osservi che $j \le \lfloor \log_b n \rfloor$ implica $b^j/n \le 1$. Il limite $f(n) = O(n^{\log_b a})$ implica che esiste una costante $c > 0$ tale che per n_j sufficientemente grande, vale

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

infatti $c(1 + b/(b-1))^{\log_b a}$ è una costante; quindi il caso 2 è dimostrato. La prova del caso 1 è quasi identica. La chiave è di dimostrare il limite $f(n_j) = O(n^{\log_b a - \varepsilon})$ con una prova simile a quella del corrispondente caso 2, malgrado i calcoli algebrici siano più complicati.

Sono stati così dimostrati i limiti superiori del teorema principale per qualunque intero n. Per i limiti inferiori la dimostrazione è analoga.

Esercizi

* **4.4-1** Dare un'espressione semplice e precisa per n_j dell'equazione (4.12) per il caso in cui b sia un intero positivo invece che un qualunque numero reale.

* **4.4-2** Mostrare che se $f(n) = \Theta(n^{\log_b a} \lg^k n)$, dove $k \ge 0$, allora la ricorrenza principale ha come soluzione $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. Per semplicità, limitarsi all'analisi di potenze esatte di b.

* **4.4-3** Mostrare che le ipotesi del caso 3 del teorema principale sono eccessive, nel senso che la condizione di regolarità $af(n/b) \le cf(n)$ per qualche costante $c < 1$, implica che esiste una costante $\varepsilon > 0$ tale che $f(n) = \Omega(n^{\log_b a + \varepsilon})$.

Problemi

4-1 *Esempi di ricorrenze*

Dare limiti inferiori e superiori per $T(n)$ per ognuna delle seguenti ricorrenze, assumendo che $T(n)$ sia costante per $n \le 2$. Definire i limiti più stretti possibile e giustificare le risposte.

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 *Trovare l'intero mancante*

Un array $A[1 \,..\, n]$ contiene tutti gli interi da 0 a n tranne uno. Sarebbe facile determinare l'intero mancante in tempo $O(n)$ usando un array ausiliario $B[0 \,..\, n]$ per registrare quali numeri appaiono in A. In questo problema, però, non si può accedere a un intero elemento di A con una singola operazione, infatti gli elementi di A sono rappresentati in binario, e la sola operazione permessa per accedervi è "prendi il j-esimo bit di $A[i]$", che richiede un tempo costante.

Mostrare che se si usa solo questa operazione, si può ancora determinare l'intero mancante in tempo $O(n)$.

4-3 *Costo del passaggio dei parametri*

In tutto il libro, si assume che il passaggio dei parametri durante le chiamate di procedure richieda tempo costante, anche se è stato passato un array di N elementi. Quest'ipotesi è valida in molti sistemi perché in realtà viene passato un puntatore all'array e non l'array stesso. Questo problema esamina le implicazioni di tre strategie di passaggio di parametri:

1. Un array è passato tramite il suo puntatore. Tempo richiesto $\Theta(1)$.
2. Un array è passato per valore. Tempo richiesto $\Theta(N)$, dove N è la dimensione dell'array.
3. Solo il sottoarray che potrebbe essere necessario alla procedura chiamata è passato per valore. Tempo richiesto $\Theta(q - p + 1)$ dove $A[p \,..\, q]$ è la porzione di array passata.

a. Considerare l'algoritmo di ricerca binaria ricorsiva per trovare un numero in un array ordinato (si veda l'Esercizio 1.3-5). Dare le ricorrenze del tempo di esecuzione della ricerca binaria nel caso peggiore quando gli array sono passati usando ognuno dei tre metodi descritti prima e dare buoni limiti superiori alle soluzioni delle ricorrenze. Indicare con N la dimensione del problema originale e con n la dimensione di un sottoproblema.

b. Ripetere la parte (a) per l'algoritmo MERGE-SORT del paragrafo 1.3.1.

4-4 *Altri esempi di ricorrenze*

Dare limiti inferiori e superiori per ognuna delle seguenti ricorrenze, assumendo che $T(n)$ sia costante per $n \le 8$. Definire i limiti più stretti possibile e giustificare le risposte.

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 *Estensione dalle potenze di un intero a tutti i reali*

Spesso si riesce a limitare una ricorrenza $T(n)$ per potenze esatte di una costante intera b. Questo problema fornisce condizioni sufficienti per estendere il limite a tutti i reali $n > 0$.

a. Siano $T(n)$ e $h(n)$ due funzioni monotone crescenti, e si supponga che $T(n) \le h(n)$ quando n è una potenza esatta di una costante $b > 1$. Si supponga inoltre che $h(n)$ sia "lentamente crescente" nel senso che $h(n) = O(h(n/b))$. Provare che $T(n) = O(h(n))$.

b. Si supponga di avere la ricorrenza $T(n) = aT(n/b) + f(n)$, dove $a \ge 1$, $b > 1$ e $f(n)$ è monotona crescente. Supporre inoltre che le condizioni iniziali siano date da $T(n) = g(n)$ per $n \le n_0$, dove $g(n)$ è monotona crescente e $g(n_0) \le aT(n_0/b) + f(n_0)$. Dimostrare che $T(n)$ è monotona crescente.

c. Semplificare la dimostrazione del teorema principale per il caso in cui $f(n)$ sia monotona e "lentamente crescente". Usare il lemma 4.4.

4-6 *Numeri di Fibonacci*

Questo problema considera alcune proprietà dei numeri di Fibonacci, che sono definiti dalla ricorrenza (2.13). Si userà la tecnica delle funzioni generatrici per risolvere la ricorrenza di Fibonacci. La ***funzione generatrice*** (o ***serie di potenze formali***) $\mathcal{F}$ è definita come

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

a. Mostrare che $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

b. Mostrare che

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1-\phi z)(1-\hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi} z}\right),\end{aligned}$$

dove

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots$$

e

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\ldots .$$

c. Mostrare che

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

d. Dimostrare che $F_i = \phi^i/\sqrt{5}$ per $i > 0$, arrotondato all'intero più vicino. (*Suggerimento*: $|\hat{\phi}| < 1$.)

e. Dimostrare che $F_{i+2} \ge \phi^i$ per $i \ge 0$.

4-7 *Controllo di chip VLSI*

Il professor Diogene ha n chip VLSI[1] che in linea di principio sono in grado di controllarsi l'un l'altro. Il dispositivo di controllo del professore verifica due chip alla volta. Quando il dispositivo è attivato, ogni chip verifica l'altro e comunica se è buono o guasto. Un chip buono comunica sempre in modo sicuro se l'altro chip è funzionante o guasto, ma la risposta di un chip guasto non può essere affidabile. Pertanto, le quattro possibili risposte risultanti da un controllo sono le seguenti:

Il chip A dice	Il chip B dice	Conclusioni
B è funzionante	*A* è funzionante	Entrambi sono funzionanti, o entrambi sono guasti
B è funzionante	*A* è guasto	almeno uno è guasto
B è guasto	*A* è funzionante	almeno uno è guasto
B è guasto	*A* è guasto	almeno uno è guasto

a. Mostrare che se più di $n/2$ chip sono guasti, il professore non può inconfutabilmente determinare quali sono i funzionanti usando una qualunque strategia basata su questo tipo di controllo a coppie. Assumere che i chip guasti possano cospirare per imbrogliare il professore.

b. Considerare il problema di trovare un singolo chip funzionante tra n chip, assumendo che più di $n/2$ siano funzionanti. Mostrare che $\lfloor n/2 \rfloor$ controlli a coppie sono sufficienti a ridurre il problema ad uno di dimensione pressoché dimezzata.

c. Mostrare che i chip funzionanti possono essere identificati con $\Theta(n)$ controlli a coppie, assumendo che più di $n/2$ sono funzionanti. Dare e risolvere la ricorrenza che descrive il numero di controlli.

[1] VLSI sta per "very-large-scale integration", cioè *integrazione su larghissima scala*, che è una tecnologia di integrazione di circuiti su piastrine di silicio (chip) usata oggi per fabbricare la maggior parte dei microprocessori.

Note al capitolo

Le ricorrenze furono studiate fin dal 1202 da L. Fibonacci, dal quale i numeri di Fibonacci hanno preso il nome. A. De Moivre introdusse il metodo delle funzioni generatrici (si veda il Problema 4-6) per risolvere le ricorrenze. Il metodo principale è adattato da Bentley, Haken e Saxe [26], che forniscono il metodo esteso presentato nell'Esercizio 4.4-2. Knuth [121] e Liu [140] mostrano come risolvere le ricorrenze lineari usando il metodo delle funzioni generatrici. Il lavoro di Purdom e Brown [164] contiene una discussione estesa sulla risoluzione delle ricorrenze.

Insiemi e affini

5

Nei capitoli precedenti sono stati introdotti elementi di analisi. Questo capitolo presenta una panoramica delle definizioni, notazioni e proprietà elementari di insiemi, relazioni, funzioni, grafi ed alberi. Il lettore già preparato su questo materiale può limitarsi a sfogliare il capitolo.

5.1 Insiemi

Un ***insieme*** è una collezione di oggetti distinguibili, chiamati ***membri o elementi***. Se un oggetto x è un elemento di un insieme S, si scrive $x \in S$ (si legge "x è un elemento di S", o più brevemente, "x è in S"); se x non è un elemento di S, si scrive $x \notin S$. Si può descrivere un insieme attraverso l'esplicito elenco dei suoi elementi scritti tra parentesi graffe; per esempio, un insieme S che contiene esattamente i numeri 1, 2 e 3, può essere definito scrivendo $S = \{1, 2, 3\}$. Inoltre, dato che 2 è un elemento di S, si può scrivere $2 \in S$, e dato che 4 non è un elemento di S si può scrivere $4 \notin S$. Un insieme non può contenere lo stesso oggetto più di una volta ed i suoi elementi non sono ordinati. Due insiemi A e B sono ***uguali*** se essi contengono gli stessi elementi: in tal caso si scrive $A = B$; per esempio, $\{1, 2, 3, 1\} = \{1, 2, 3\} = \{3, 2, 1\}$.

Nel libro si adottano le seguenti notazioni speciali per alcuni insiemi che saranno usati più frequentemente:

- $\varnothing$ denota l'***insieme vuoto***, cioè l'insieme che non contiene alcun elemento.
- $\mathbf{Z}$ denota l'insieme degli ***interi***, cioè l'insieme $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.
- $\mathbf{R}$ denota l'insieme dei ***numeri reali***.
- $\mathbf{N}$ denota l'insieme dei ***numeri naturali***, cioè l'insieme $\{0, 1, 2, \ldots\}$[1].

Se tutti gli elementi di un insieme A sono contenuti in un insieme B, cioè, se $x \in A$ implica $x \in B$, allora si scrive $A \subseteq B$ e si dice che A è un ***sottoinsieme*** di B; un insieme A è ***un sottoinsieme proprio*** di B, scritto $A \subset B$, se $A \subseteq B$ ma $A \neq B$. (Alcuni autori usano il simbolo "$\subset$" per indicare la relazione di sottoinsieme, piuttosto che la relazione di sottoinsieme proprio). Per ogni insieme A vale $A \subseteq A$; per due insiemi A e B, si ha $A = B$ se e solo se $A \subseteq B$ e $B \subseteq A$; per tre insiemi A, B e C, se $A \subseteq B$ e $B \subseteq C$, allora $A \subseteq C$; per ogni insieme A si ha che $\varnothing \subseteq A$.

Si possono descrivere insiemi mediante altri insiemi: dato un insieme A, si può definire un insieme $B \subseteq A$ stabilendo una proprietà che individui gli elementi di B. Per esempio, si può definire l'insieme dei numeri interi pari con $\{x : x \in \mathbf{Z} \text{ e } x/2 \text{ è un intero}\}$; in questa notazione i ":" sono letti come "tale che". (Alcuni autori usano una barra verticale al posto dei ":".)

[1] Alcuni autori iniziano i numeri naturali con 1 anziché con 0. La tendenza moderna sembra essere quella di iniziare con 0.

Dati due insiemi A e B, si possono definire nuovi insiemi anche applicando le seguenti ***operazioni su insiemi***:

- L'***intersezione*** degli insiemi A e B è l'insieme
  $A \cap B = \{x : x \in A \text{ e } x \in B\}$.
- L'***unione*** degli insiemi A e B è l'insieme
  $A \cup B = \{x : x \in A \text{ o } x \in B\}$.
- La ***differenza*** tra due insiemi A e B è l'insieme
  $A - B = \{x : x \in A \text{ e } x \notin B\}$.

Le operazioni sugli insiemi rispettano le seguenti proprietà o leggi.

Proprietà dell'insieme vuoto:

$A \cap \varnothing = \varnothing$,

$A \cup \varnothing = A$.

Idempotenza:

$A \cap A = A$.

$A \cup A = A$.

Commutatività:

$A \cap B = B \cap A$,

$A \cup B = B \cup A$.

Associatività:

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

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

Distributività:

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

$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. (5.1)

Assorbimento:

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

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

Leggi di DeMorgan:

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

$A - (B \cup C) = (A - B) \cap (A - C)$. (5.2)

La prima legge di DeMorgan è illustrata nella figura 5.1 usando un ***diagramma di Venn*** che è una rappresentazione grafica nella quale gli insiemi sono raffigurati come regioni del piano.

Spesso, gli insiemi che si stanno considerando sono sottoinsiemi di un insieme U più grande chiamato ***universo***; per esempio, se si stanno considerando diversi insiemi costituiti solo da numeri interi, allora l'insieme **Z** è un universo appropriato. Dato un universo U, si definisce il ***complemento*** di un insieme A come $\overline{A} = U - A$; per ogni insieme $A \subseteq U$, si hanno le seguenti proprietà:

Figura 5.1 *Un diagramma di Venn che illustra la prima legge di DeMorgan* (5.2). *Ciascun insieme A, B e C è rappresentato nel piano con un cerchio.*

$\overline{\overline{A}} = A$,

$A \cap \overline{A} = \varnothing$,

$A \cup \overline{A} = U$.

Le leggi di DeMorgan (5.2) possono essere riscritte con i complementi: per ogni coppia di insiemi $A, B \subseteq U$, si ha

$\overline{A \cap B} = \overline{A} \cup \overline{B}$,

$\overline{A \cup B} = \overline{A} \cap \overline{B}$.

Due insiemi A e B sono ***disgiunti*** se non hanno alcun elemento in comune, cioè, se $A \cap B = \varnothing$.

Una collezione $\mathcal{S} = \{S_i\}$ di insiemi non vuoti, forma ***una partizione*** di un insieme S se:

- gli insiemi sono ***a coppie disgiunti***, cioè, $S_i, S_j \in \mathcal{S}$ e $i \neq j$ implica $S_i \cap S_j = \varnothing$, e
- la loro unione è S, cioè

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

In altre parole $\mathcal{S}$ forma una partizione di S se ciascun elemento di S appare esattamente in un $S_i \in \mathcal{S}$.

Il numero degli elementi di un insieme è chiamato ***cardinalità*** (***o dimensione***) dell'insieme ed è denotato da $|S|$. Due insiemi hanno la stessa cardinalità se i loro elementi possono essere messi in corrispondenza biunivoca; la cardinalità dell'insieme vuoto è $|\varnothing| = 0$. Se la cardinalità di un insieme è un numero naturale, allora l'insieme si dice ***finito***, altrimenti si dice ***infinito***. Un insieme infinito che può essere messo in corrispondenza biunivoca con l'insieme **N** dei numeri naturali, si dice ***infinito numerabile***, altrimenti si dice ***non numerabile***. Per esempio, l'insieme **Z** degli interi è numerabile, mentre l'insieme **R** dei reali è non numerabile.

Per ogni coppia A e B di insiemi finiti, vale l'identità

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

da cui si può concludere che

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

Se A e B sono disgiunti, allora $|A \cap B| = 0$ e quindi $|A \cup B| = |A| + |B|$. Se $A \subseteq B$ allora $|A| \leq |B|$.

Un insieme finito di n elementi è chiamato talvolta ***n-insieme***; un 1-insieme è chiamato ***singoletto***; un sottoinsieme di k elementi è chiamato ***k-sottoinsieme***.

L'insieme di tutti i sottoinsiemi di un insieme S, inclusi l'insieme vuoto e l'insieme S stesso, è denotato da 2^S ed è chiamato ***l'insieme potenza*** di S; per esempio, $2^{\{a,b\}} = \{\varnothing, \{a\}, \{b\}, \{a, b\}\}$. L'insieme potenza di un insieme finito S ha cardinalità $2^{|S|}$.

Spesso si considerano strutture simili ad insiemi, nelle quali gli elementi sono ordinati; per esempio, una ***coppia ordinata*** di due elementi a e b è denotata con (a, b) e può essere formalmente definita come l'insieme $(a, b) = \{a, \{a, b\}\}$; quindi la coppia ordinata (a, b) *non* è uguale alla coppia ordinata (b, a).

Il ***prodotto cartesiano*** di due insiemi A e B, denotato da $A \times B$, è l'insieme di tutte le coppie ordinate tali che il primo elemento della coppia è un elemento di A ed il secondo è un elemento di B; più formalmente

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

Per esempio, $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$. Quando A e B sono insiemi finiti, la cardinalità del loro prodotto cartesiano è

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

Il prodotto cartesiano di n insiemi $A_1, A_2, \ldots, A_n$ è l'insieme di ***n–uple***

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

la cui cardinalità è

$$|A_1 \times A_2 \times \ldots \times A_n| = |A_1| \cdot |A_2| \cdot \ldots |A_n|$$

se tutti gli insiemi sono finiti. Si può denotare il prodotto cartesiano ripetuto n volte di un singolo insieme A con l'insieme

$$A^n = A \times A \times \ldots \times A,$$

la cui cardinalità è $|A^n| = |A|^n$ se A è finito. Una n-upla può essere anche vista come una sequenza finita di lunghezza n (si veda il Paragrafo 5.3).

Esercizi

5.1-1 Disegnare i diagrammi di Venn che illustrano la prima proprietà distributiva (5.1).

5.1-2 Dimostrare la generalizzazione delle leggi di DeMorgan per qualsiasi collezione finita di insiemi

$$\overline{A_1 \cap A_2 \cap \ldots \cap A_n} = \overline{A_1} \cap \overline{A_2} \cap \ldots \cap \overline{A_n}$$
$$\overline{A_1 \cup A_2 \cup \ldots \cup A_n} = \overline{A_1} \cup \overline{A_2} \cup \ldots \cup \overline{A_n}$$

* *5.1-3* Dimostrare la generalizzazione dell'equazione (5.3), che è chiamato ***principio di inclusione ed esclusione***:

$$\begin{aligned} |A_1 \cup A_2 \cup \ldots \cup A_n| = & \\ & |A_1| + |A_2| + \ldots + |A_n| \\ & -|A_1 \cap A_2| - |A_1 \cap A_3| - \ldots && \text{(per tutte le coppie)} \\ & +|A_1 \cap A_2 \cap A_3| + \ldots && \text{(per tutte le triple)} \\ & \vdots \\ & +(-1)^{n-1}|A_1 \cap A_2 \cap \ldots \cap A_n|. \end{aligned}$$

UNIVERSITA' DI NAPOLI FEDERICO II BIBLIOTECA CENTRALE

5.1-4 Mostrare che l'insieme dei numeri dispari è numerabile.

5.1-5 Mostrare che, per ogni insieme S finito, l'insieme potenza 2^S ha $2^{|S|}$ elementi (cioè, ci sono $2^{|S|}$ sottoinsiemi distinti di S).

5.1-6 Dare una definizione induttiva di n-upla, estendendo la definizione di coppia ordinata fornita dalla teoria degli insiemi.

5.2 Relazioni

Una ***relazione binaria*** R su due insiemi A e B è un sottoinsieme del prodotto cartesiano $A \times B$. Se $(a, b) \in R$, talvolta sarà scritto $a\,R\,b$. Quando si dice che R è una relazione binaria su A, si intende che R è un sottoinsieme di $A \times A$. Per esempio, la relazione "minore di" sui numeri naturali è l'insieme $\{(a, b) : a, b \in \mathbf{N} \text{ e } a < b\}$. Una relazione n-aria sugli insiemi $A_1, A_2, \ldots, A_n$ è un sottoinsieme di $A_1 \times A_2 \times \ldots \times A_n$.

Una relazione binaria $R \subseteq A \times A$ è ***riflessiva*** se

$$a\,R\,a$$

per tutti gli $a \in A$. Per esempio "=" e "≤" sono relazioni riflessive su $\mathbf{N}$, mentre "<" non lo è. La relazione R è ***simmetrica*** se

$a\,R\,b$ implica $b\,R\,a$

per tutti gli $a, b \in A$. Per esempio "=" è simmetrica, mentre "<" e "≤" non lo sono. La relazione R è ***transitiva*** se

$a\,R\,b$ e $b\,R\,c$ implicano $a\,R\,c$

per tutti gli $a, b, c \in A$. Per esempio, le relazioni "≤", "<" e "=" sono transitive, mentre la relazione $R = \{(a, b) : a, b \in \mathbf{N} \text{ e } a = b - 1\}$ non lo è, in quanto $3\,R\,4$ e $4\,R\,5$ non implicano $3\,R\,5$.

Una relazione che sia riflessiva, simmetrica e transitiva è una ***relazione di equivalenza***. Per esempio, "=" è una relazione di equivalenza sui numeri naturali, mentre "<" non lo è. Se R è una relazione di equivalenza su un insieme A, per $a \in A$ la ***classe di equivalenza*** di a è l'insieme $[a] = \{b : b \in A \text{ e } a\,R\,b\}$, cioè l'insieme di tutti gli elementi equivalenti ad a. Per esempio, se si definisce $R = \{(a, b) : a, b \in \mathbf{N} \text{ e } a + b \text{ è un numero pari}\}$, allora R è una relazione di equivalenza, in quanto $a + a$ è pari (riflessiva), $a + b$ pari implica $b + a$ pari (simmetrica) e $a + b$ pari e $b + c$ pari implicano $a + c$ pari (transitiva). La classe di equivalenza di 4 è $[4] = \{0, 2, 4, 6, \ldots\}$ e la classe di equivalenza di 3 è $[3] = \{1, 3, 5, 7, \ldots\}$. Un teorema base delle classi di equivalenza è il seguente.

Teorema 5.1 (***Una relazione di equivalenza coincide con una partizione***)

Le classi di equivalenza di una qualunque relazione di equivalenza R su un insieme A formano una partizione di A e qualsiasi partizione di A determina una relazione di equivalenza su A in cui le classi di equivalenza sono gli insiemi della partizione.

Dimostrazione. Per la prima parte della dimostrazione si deve mostrare che le classi di equivalenza di R sono non vuote, sono due a due disgiunte e la loro unione è A. Dato che R è riflessiva, $a \in [a]$ e quindi le classi di equivalenza non sono vuote; inoltre, dato che ogni elemento $a \in A$ appartiene alla classe di equivalenza $[a]$, l'unione delle classi di equivalenza è A. Rimane da mostrare che le classi di equivalenza sono a coppie disgiunte, cioè, se due classi di equivalenza $[a]$ e $[b]$ hanno un elemento c in comune, allora esse sono di fatto lo stesso insieme. Partendo da $a\,R\,c$ e $b\,R\,c$, per simmetria e transitività si ha che $a\,R\,b$. Di conseguenza, per ogni generico elemento $x \in [a]$, $x\,R\,a$ implica $x\,R\,b$ e perciò $[a] \subseteq [b]$. Nello stesso modo si verifica che $[b] \subseteq [a]$ per cui $[a] = [b]$.

Per la seconda parte della dimostrazione, sia $\mathcal{A} = \{A_i\}$ una partizione di A e si definisca $R = \{(a, b)$: esiste i tale che $a \in A_i$ e $b \in A_i\}$; si stabilisce che R è una relazione di equivalenza su A. La riflessività è soddisfatta, dato che $a \in A_i$ implica $a\,R\,a$; la simmetria è soddisfatta, perché se $a\,R\,b$, allora a e b sono nello stesso insieme A_i, e quindi $b\,R\,a$; se $a\,R\,b$ e $b\,R\,c$, allora tutti e tre gli elementi sono nello stesso insieme e quindi $a\,R\,c$ per cui anche la transitività è soddisfatta. Per vedere che gli insiemi della partizione sono classi di equivalenza di R, si osservi che se $a \in A_i$ allora $x \in [a]$ implica $x \in A_i$, e $x \in A_i$ implica $x \in [a]$. ■

Una relazione binaria R *su* un insieme A è ***antisimmetrica*** se

$a\,R\,b$ e $b\,R\,a$ implicano $a = b$.

Per esempio, la relazione "$\leq$" sui numeri naturali è antisimmetrica in quanto $a \leq b$ e $b \leq a$ implicano che $a = b$. Una relazione che sia riflessiva, antisimmetrica e transitiva è un ***ordinamento parziale*** e l'insieme su cui l'ordinamento parziale è definito si dice ***insieme parzialmente ordinato***. Per esempio, la relazione "discendente di" è un ordinamento parziale sull'insieme di tutte le persone (se si considerano gli individui anche come discendenti di se stessi).

In un insieme A parzialmente ordinato, può non esistere un singolo elemento x "massimo", tale che $y\,R\,x$ per tutti gli $y \in A$; ci saranno, invece, diversi elementi ***massimali*** x tali che per nessun elemento $y \in A$ si avrà $x\,R\,y$. Per esempio, in un insieme di scatole di diversa grandezza ci potrebbero essere diverse scatole massimali che non entrano in nessun'altra scatola e tuttavia non c'è una singola scatola "massima" in cui qualsiasi altra scatola potrà entrare.

Un ordinamento parziale R su un insieme A è un ***ordinamento totale*** o ***ordinamento lineare*** se per tutti gli $a, b \in A$, si ha che $a\,R\,b$ o $b\,R\,a$, cioè se ogni coppia di elementi di A può essere messa in relazione secondo R. Per esempio, la relazione "$\leq$" è totale sui numeri naturali, mentre la relazione "discendente di" non è un ordinamento totale sull'insieme delle persone in quanto ci possono essere individui che non discendono l'uno dall'altro.

Esercizi

5.2-1 Provare che la relazione di inclusione insiemistica "$\subseteq$" su tutti i sottoinsiemi di $\mathbb{Z}$ è un ordinamento parziale ma non è un ordinamento totale.

5.2-2 Mostrare che per qualunque intero positivo n, la relazione "equivalente modulo n" è una relazione di equivalenza sugli interi. (Si dice che $a \equiv b \pmod{n}$ se esiste un intero q tale che $a - b = qn$). In quali classi di equivalenza questa relazione partiziona gli interi?

5.2-3 Dare esempi di relazioni che sono:

a. riflessive e simmetriche ma non transitive,

b. riflessive e transitive ma non simmetriche,

c. simmetriche e transitive ma non riflessive.

5.2-4 Sia S un insieme finito ed R una relazione di equivalenza su $S \times S$. Verificare che R è anche antisimmetrica, allora le classi di equivalenza di R sono singoletti.

5.2-5 Il professor Narciso afferma che se una relazione R è simmetrica e transitiva allora è anche riflessiva. Egli dà la seguente dimostrazione: per simmetria $a\,R\,b$ implica $b\,R\,a$, e quindi la transitività implica $a\,R\,a$. La dimostrazione è corretta?

5.3 Funzioni

Dati due insiemi A e B, una ***funzione*** f è una relazione binaria su $A \times B$ tale che, per tutti gli $a \in A$, esiste uno e un solo $b \in B$ tale che $(a, b) \in f$. L'insieme A è chiamato il ***dominio*** di f e l'insieme B è chiamato il ***codominio*** di f. Talvolta si scrive $f : A \to B$ e, se $(a, b) \in f$, si scrive $b = f(a)$ dato che b è univocamente determinato con la scelta di a.

Intuitivamente, la funzione f assegna un elemento di B ad ogni elemento di A: nessun elemento di A è assegnato a due distinti elementi di B, ma lo stesso elemento di B può essere assegnato a due distinti elementi di A. Per esempio la relazione binaria

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

è una funzione $f : \mathbf{N} \to \{0, 1\}$, dato che per ogni numero naturale a c'è esattamente un solo valore b in $\{0, 1\}$ tale che $b = a \bmod 2$. Per questo esempio $0 = f(0)$, $1 = f(1)$, $0 = f(2)$ ecc. Viceversa, la relazione binaria

$g = \{(a, b) : a \in \mathbf{N} \text{ e } a + b \text{ è pari}\}$

non è una funzione dato che $(1, 3)$ e $(1, 5)$ sono entrambe in g: di conseguenza, scegliendo $a = 1$, non c'è esattamente un unico b tale che $(a, b) \in g$.

Data una funzione $f : A \to B$, se $f(a) = b$ si dice che a è l'***argomento*** di f e b è il ***valore*** di f in a. Si può definire una funzione specificando il suo valore per ogni elemento del dominio: per esempio, si potrebbe definire $f(n) = 2n$ per $n \in \mathbf{N}$, che significa $f = \{(n, 2n) : n \in \mathbf{N}\}$. Due funzioni f e g sono ***uguali*** se hanno lo stesso dominio, lo stesso codominio e se, per tutte le a del dominio, $f(a) = g(a)$.

Una ***sequenza finita*** di lunghezza n è una funzione f il cui dominio è l'insieme $\{0, 1, \ldots, n - 1\}$: spesso si rappresenta una sequenza finita elencando i suoi valori: $\langle f(0), f(1), \ldots, f(n - 1)\rangle$. Una ***sequenza infinita*** è una funzione il cui dominio è l'insieme $\mathbf{N}$ dei numeri naturali: per esempio, la sequenza di Fibonacci definita in (2.13), è la sequenza infinita $\langle 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots\rangle$.

Quando il dominio di una funzione f è un prodotto cartesiano, spesso si omettono le parentesi che racchiudono l'argomento di f: per esempio, se $f : A_1 \times A_2 \times \ldots \times A_n \to B$, si scrive $b = f(a_1, a_2, \ldots, a_n)$, invece di $b = f((a_1, a_2, \ldots, a_n))$. Inoltre, ciascun elemento a_i è chiamato ***argomento*** della funzione f, benché tecnicamente il (singolo) argomento di f sia la n-upla $(a_1, a_2, \ldots, a_n)$.

Se $f: A \to B$ è una funzione e $b = f(a)$, allora talvolta si dice che b è l'***immagine*** di a secondo f; l'immagine di un insieme $A' \subseteq A$ secondo f è definita da

$$f(A') = \{b \in B : b = f(a) \text{ per qualche } a \in A'\}.$$

Il ***rango*** di f è l'immagine del suo dominio, cioè $f(A)$. Per esempio, il rango della funzione $f: \mathbf{N} \to \mathbf{N}$ definita da $f(n) = 2n$ è $f(\mathbf{N}) = \{m : m = 2n \text{ per qualche } n \in \mathbf{N}\}$.

Una funzione è ***surgettiva*** se il suo rango coincide con il suo codominio. Per esempio, la funzione $f(n) = \lfloor n/2 \rfloor$ è una funzione surgettiva da $\mathbf{N}$ a $\mathbf{N}$, dato che ogni elemento di $\mathbf{N}$ appare come valore di f per qualche argomento. Viceversa, la funzione $f(n) = 2n$ non è una funzione surgettiva da $\mathbf{N}$ a $\mathbf{N}$, dato che nessun argomento di f può produrre 3 come suo valore. La funzione $f(n) = 2n$ è, tuttavia, una funzione surgettiva dai numeri naturali ai numeri pari. Una funzione surgettiva $f: A \to B$ è talvolta descritta come una corrispondenza di A ***su*** B. Quando si dice che f è *su* qualche insieme, allora si intende che f è surgettiva.

Una funzione $f: A \to B$ è ***iniettiva*** se ad argomenti distinti di f corrispondono valori diversi, cioè, se $a \neq a'$ implica $f(a) \neq f(a')$. Per esempio, la funzione $f(n) = 2n$ è una funzione iniettiva da $\mathbf{N}$ a $\mathbf{N}$, dato che un numero pari b è immagine, per f, di un solo elemento del dominio e precisamente $b/2$. La funzione $f(n) = \lfloor n/2 \rfloor$ non è iniettiva in quanto il valore 1 corrisponde a due argomenti: 2 e 3. Una funzione iniettiva è talvolta chiamata una funzione ***uno-a-uno***.

Una funzione $f: A \to B$ è ***biunivoca*** se è iniettiva e surgettiva; per esempio, la funzione $f(n) = (-1)^n \lceil n/2 \rceil$ è biunivoca da $\mathbf{N}$ a $\mathbf{Z}$:

$$\begin{aligned} 0 &\to 0, \\ 1 &\to -1, \\ 2 &\to 1, \\ 3 &\to -2, \\ 4 &\to 2, \\ &\vdots \end{aligned}$$

La funzione è iniettiva dato che nessun elemento di $\mathbf{Z}$ è l'immagine di più di un elemento di $\mathbf{N}$; è surgettiva dato che tutti gli elementi di $\mathbf{Z}$ sono immagine di qualche elemento di $\mathbf{N}$, quindi la funzione è biunivoca. Una funzione biunivoca è chiamata talvolta ***corrispondenza uno-a uno***, in quanto accoppia gli elementi del dominio e del codominio. Una funzione biunivoca da un insieme A su se stesso è talvolta chiamata una ***permutazione***.

Quando una funzione f è biunivoca, la sua ***inversa*** f^{-1} è definita come

$$f^{-1}(b) = a \text{ se e solo se } f(a) = b.$$

Per esempio, l'inversa della funzione $f(n) = (-1)^n \lceil n/2 \rceil$ è

$$f^{-1}(m) = \begin{cases} 2m & \text{se } m \geq 0, \\ -2m - 1 & \text{se } m < 0. \end{cases}$$

Esercizi

5.3-1 Siano A e B due insiemi finiti e sia $f: A \to B$ una funzione. Mostrare che:

a. se f è iniettiva, allora $|A| \leq |B|$;

b. se f è surgettiva, allora $|A| \geq |B|$.

5.3-2 La funzione $f(x) = x + 1$ è biunivoca quando il dominio ed il codominio sono $\mathbf{N}$? È biunivoca quando il dominio ed il codominio sono $\mathbf{Z}$?

5.3-3 Dare una definizione naturale per l'inversa di una relazione binaria, in modo tale che, se una relazione è in realtà una funzione biunivoca, allora la sua inversa relazionale coincida con l'inversa funzionale.

* *5.3-4* Dare una funzione biunivoca da $\mathbf{Z}$ a $\mathbf{Z} \times \mathbf{Z}$.

5.4 Grafi

Questo paragrafo presenta due tipi di grafi: orientati e non orientati. Il lettore troverà che certe definizioni presenti in letteratura differiscono da quelle date in questo paragrafo, anche se, per la maggior parte, queste differenze sono minime. Il paragrafo 23.1 mostra come i grafi possono essere rappresentati nella memoria di un calcolatore.

Un ***grafo orientato*** (o ***diretto*** o ***di-grafo***) G è una coppia (V, E), dove V è un insieme finito ed E è una relazione binaria su V. L'insieme V è chiamato l'***insieme dei vertici*** di G ed i suoi elementi sono chiamati ***vertici***. L'insieme E è chiamato l'***insieme degli archi*** di G ed i suoi elementi sono chiamati ***archi***. La figura 5.2 (a) è una rappresentazione di un grafo orientato con insieme dei vertici $\{1, 2, 3, 4, 5, 6\}$. Nella figura i vertici sono disegnati con cerchi e gli archi sono disegnati con frecce. Si noti che sono possibili ***cappi***, archi da un vertice a se stesso.

In un ***grafo non orientato*** $G = (V, E)$, l'insieme degli archi E è costituito da coppie *non ordinate* di vertici, piuttosto che da coppie ordinate; cioè, un arco è un insieme $\{u, v\}$, dove $u, v \in V$ e $u \neq v$. Per convenzione si usa la notazione (u, v) per un arco, piuttosto che la notazione insiemistica $\{u, v\}$; inoltre (u, v) e (v, u) sono considerati essere lo stesso arco. In un grafo non orientato i cappi sono proibiti, per cui ogni arco consiste esattamente di due vertici distinti. La figura 5.2 (b) è una rappresentazione di un grafo non orientato con insieme dei vertici $\{1, 2, 3, 4, 5, 6\}$.

Molte definizioni per grafi orientati e non orientati coincidono, benché certi termini abbiano un significato leggermente diverso. Se (u, v) è un arco di un grafo orientato $G = (V, E)$, si dice che (u, v) è ***incidente*** o ***esce dal*** vertice u ed è ***incidente*** o ***entra nel*** vertice v. Per esempio, gli archi che escono dal vertice 2 nella figura 5.2(a) sono $(2, 2)$, $(2, 4)$ e $(2, 5)$; mentre gli archi che entrano nel vertice 2 sono $(1, 2)$ e $(2, 2)$. Se (u, v) è un arco di un grafo non orientato $G = (V, E)$, si dice che (u, v) è ***incidente sui*** vertici u e v. Nella figura 5.2(b), gli archi incidenti sul vertice 2 sono $(1, 2)$ e $(2, 5)$.

Se (u, v) è un arco di un grafo $G = (V, E)$, si dice che il vertice v è ***adiacente*** al vertice u. Quando il grafo è non orientato la relazione di adiacenza è simmetrica, mentre quando è orientato la relazione non è necessariamente simmetrica. Se v è adiacente a u in un grafo orientato talvolta si scrive $u \to v$. Nelle parti (a) e (b) della figura 5.2, il vertice 2 è adiacente al vertice 1, infatti l'arco $(1,2)$ è presente in entrambi i grafi. Il vertice 1 *non* è adiacente al vertice 2 nella figura 5.2 (a), poiché l'arco $(2,1)$ non appartiene al grafo.

Figura 5.2 *Grafi orientati e non orientati.* (a) *Un grafo orientato* $G = (V, E)$, *dove* $V = \{1, 2, 3, 4, 5, 6\}$ *ed* $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. *L'arco* $(2, 2)$ *è un cappio.* (b) *Un grafo non orientato* $G = (V, E)$, *dove* $V = \{1, 2, 3, 4, 5, 6\}$ *ed* $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. *Il vertice* 4 *è isolato.* (c) *Il sottografo del grafo della parte (a), indotto dall'insieme di vertici* $\{1, 2, 3, 6\}$.

Il ***grado*** di un vertice in un grafo non orientato è il numero di archi incidenti su di esso. Per esempio, il vertice 2 nella figura 5.2 (b) ha grado 2. In un grafo orientato, il ***grado uscente*** di un vertice è il numero di archi che escono da esso ed il ***grado entrante*** è il numero di archi che entrano nel vertice. Il ***grado*** di un vertice in un grafo orientato è il suo grado entrante più il suo grado uscente. Il vertice 2 nella figura 5.2 (a) ha grado entrante 2, grado uscente 3 e grado 5. Un vertice di grado 0, come il vertice 4 nella figura 5.2 (b), si dice ***isolato***.

Un ***cammino*** di ***lunghezza*** k da un vertice u ad un vertice u' in un grafo $G = (V, E)$ è una sequenza $\langle v_0, v_1, v_2, \ldots, v_k\rangle$ di vertici tale che $u = v_0$, $u' = v_k$ e $(v_{i-1}, v_i) \in E$ per $i = 1, 2, \ldots, k$. La lunghezza di un cammino è il suo numero di archi. Il cammino ***contiene*** i vertici $v_0, v_1, \ldots, v_k$ e gli archi $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. Se vi è un cammino p da u a u', si dice che u' è ***raggiungibile*** da u tramite p; ciò talvolta, se G è orientato, si scrive come $u \overset{p}{\leadsto} u'$. Un cammino è ***semplice*** se tutti i vertici sono distinti. Nella figura 5.2 (a) il cammino $\langle 1, 2, 5, 4\rangle$ è un cammino semplice di lunghezza 3. Il cammino $\langle 2, 5, 4, 5\rangle$ non è semplice.

Un ***sottocammino*** di un cammino $p = \langle v_0, v_1, \ldots, v_k\rangle$ è una sottosequenza contigua dei suoi vertici. Cioè, per qualunque $0 \le i \le j \le k$, la sottosequenza di vertici $\langle v_i, v_{i+1}, \ldots, v_j\rangle$ è un sottocammino di p.

In un grafo orientato, un cammino $\langle v_0, v_1, \ldots, v_k\rangle$ forma un ***ciclo*** se $v_0 = v_k$ e il cammino contiene almeno un arco. Il ciclo è ***semplice***, se $v_1, v_2, \ldots, v_k$ sono distinti. Un cappio è un ciclo di lunghezza 1. Due cammini $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_0\rangle$ e $\langle v'_0, v'_1, v'_2, \ldots, v'_{k-1}, v'_0\rangle$ formano lo stesso ciclo se esiste un intero j tale che $v'_i = v_{(i+j) \bmod k}$ per $i = 0, 1, \ldots, k-1$. Nella figura 5.2 (a) il cammino $\langle 1, 2, 4, 1\rangle$ forma lo stesso ciclo dei cammini $\langle 2, 4, 1, 2\rangle$ e $\langle 4, 1, 2, 4\rangle$. Questo ciclo è semplice, ma il ciclo $\langle 1, 2, 4, 5, 4, 1\rangle$ non lo è. Il ciclo $\langle 2, 2\rangle$ formato dall'arco $(2, 2)$ è un cappio. Un grafo orientato senza cappi è ***semplice***.

In un grafo non orientato, un cammino $\langle v_0, v_1, \ldots, v_k\rangle$ forma un ***ciclo (semplice)*** se $k \ge 3$, $v_0 = v_k$ e $v_1, v_2, \ldots, v_k$ sono distinti. Per esempio, nella figura 5.2 (b), il cammino $\langle 1, 2, 5, 1\rangle$ è un ciclo. Un grafo senza cicli è ***aciclico***.

Un grafo non orientato è ***connesso*** se ogni coppia di vertici è collegata con un cammino. Le ***componenti connesse*** di un grafo sono le classi di equivalenza dei vertici sotto la relazione "raggiungibile da". Il grafo nella figura 5.2 (b) ha tre componenti connesse: $\{1, 2, 5\}$, $\{3, 6\}$ e $\{4\}$. Ogni vertice in $\{1, 2, 5\}$ è raggiungibile da ogni altro vertice in $\{1, 2, 5\}$. Un grafo non orientato è connesso se ha esattamente una componente connessa, cioè se ogni vertice è raggiungibile da ogni altro vertice.

Figura 5.3 (a) *Una coppia di grafi isomorfi. I vertici del grafo in alto sono messi in corrispondenza con i vertici del grafo in basso da* $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$. (b) *I due grafi non sono isomorfi, dato che il grafo in alto ha un vertice di grado 4 mentre il grafo in basso non lo ha.*

Un grafo orientato è ***fortemente connesso*** se ogni vertice è raggiungibile da ogni altro. Le ***componenti fortemente connesse*** di un grafo sono le classi di equivalenza dei vertici sotto la relazione "mutuamente raggiungibile". Un grafo orientato è fortemente connesso se ha soltanto una componente fortemente connessa. Il grafo nella figura 5.2 (a) ha tre componenti fortemente connesse: $\{1, 2, 4, 5\}$, $\{3\}$ e $\{6\}$. Tutte le coppie di vertici in $\{1, 2, 4, 5\}$ sono mutuamente raggiungibili. I vertici $\{3, 6\}$ non formano una componente fortemente connessa, poiché il vertice 6 non può essere raggiunto dal vertice 3.

Due grafi $G = (V, E)$ e $G' = (V', E')$ sono ***isomorfi*** se esiste una funzione biunivoca $f: V \to V'$ tale che $(u, v) \in E$ se e solo se $(f(u), f(v)) \in E'$. In altre parole si possono rietichettare i vertici di G perché siano i vertici di G', mantenendo i corrispondenti archi in G e G'. La figura 5.3 (a) mostra una coppia di grafi isomorfi G e G' con rispettivi insiemi di vertici $V = \{1, 2, 3, 4, 5, 6\}$ e $V' = \{u, v, w, x, y, z\}$. La corrispondenza da V a V' data da $f(1) = u$, $f(2) = v$, $f(3) = w$, $f(4) = x$, $f(5) = y$, $f(6) = z$ è la funzione biunivoca richiesta. I grafi nella figura 5.3 (b) non sono isomorfi; infatti, sebbene entrambi i grafi abbiano 5 vertici e 7 archi, il grafo in alto ha un vertice di grado 4 che quello in basso non ha.

Si dice che un grafo $G' = (V', E')$ è un ***sottografo*** di $G = (V, E)$ se $V' \subseteq V$ ed $E' \subseteq E$. Dato un insieme $V' \subseteq V$, il sottografo di G ***indotto*** da V' è il grafo $G' = (V', E')$, dove

$$E' = \{(u, v) \in E : u, v \in V'\}.$$

Il sottografo indotto dall'insieme dei vertici $\{1, 2, 3, 6\}$ nella figura 5.2 (a) appare nella figura 5.2 (c) ed ha come insieme di archi $\{(1, 2), (2, 2), (6, 3)\}$.

Dato un grafo non orientato $G = (V, E)$, la ***versione orientata*** di G è il grafo orientato $G' = (V, E')$, dove $(u, v) \in E'$ se e solo se $(u, v) \in E$. Cioè ogni arco non orientato (u, v) in G è sostituito nella versione orientata da due archi orientati (u, v) e (v, u). Dato un grafo orientato $G = (V, E)$, la ***versione non orientata*** di G è il grafo non orientato $G' = (V, E')$, dove $(u, v) \in E'$ se e solo se $u \ne v$ e $(u, v) \in E$. Cioè la versione non orientata contiene gli archi di G "con le frecce rimosse" e senza cappi. (Poiché (u, v) e (v, u) sono lo stesso arco in un grafo non orientato, la versione non orientata di un grafo orientato lo contiene una sola volta, anche se il grafo orientato contiene entrambi gli archi (u, v) e (v, u).) In un grafo orientato $G = (V, E)$ un ***vicino*** di un vertice u è qualunque vertice che sia adiacente ad u nella versione non orientata di G. Cioè v è un vicino di u se $(u, v) \in E$ oppure $(v, u) \in E$. In un grafo non orientato, u e v sono vicini se sono adiacenti.

Alcuni tipi di grafi hanno dei nomi specifici. Un ***grafo completo*** è un grafo non orientato in cui ogni coppia di vertici è adiacente. Un ***grafo bipartito*** è un grafo non orientato $G = (V, E)$ in cui V può essere partizionato in due insiemi V_1, V_2 tali che $(u, v) \in E$ implica che o $u \in V_1$ e $v \in V_2$ oppure $u \in V_2$ e $v \in V_1$. Cioè tutti gli archi vanno da V_1 a V_2 o viceversa. Un grafo aciclico e non orientato è una ***foresta*** e un grafo connesso aciclico e non orientato è un ***albero (libero)*** (si veda il paragrafo 5.5). Spesso si usano le prime lettere del nome inglese "directed acyclic graph" (***dag***) per indicare un grafo aciclico orientato.

Vi sono due varianti dei grafi che si possono incontrare occasionalmente. Un ***multigrafo*** è come un grafo non orientato, ma può avere sia archi multipli tra i vertici che cappi. Un ***ipergrafo*** è come un grafo non orientato, ma ogni ***iperarco***, piuttosto che connettere due vertici, connette un sottoinsieme arbitrario di vertici. Molti algoritmi per grafi ordinari orientati e non orientati possono essere adattati per essere eseguiti su queste strutture simili ai grafi.

Esercizi

5.4-1 I partecipanti ad un ricevimento di facoltà si stringono la mano per salutarsi e ogni professore ricorda quante mani ha stretto. Alla fine del ricevimento il direttore del dipartimento calcola quante mani ha stretto ogni professore. Mostrare che il risultato è pari provando il seguente ***lemma delle strette di mano***: se $G = (V, E)$ è un grafo non orientato, allora

$$\sum_{v \in V} \text{grado}(v) = 2|E| .$$

5.4-2 Dimostrare che in un grafo non orientato, la lunghezza di un ciclo deve essere almeno 3.

5.4-3 Mostrare che se un grafo orientato o non orientato contiene un cammino tra due vertici u e v, allora contiene un cammino semplice tra u e v. Mostrare che se un grafo orientato contiene un ciclo, allora contiene un ciclo semplice.

5.4-4 Mostrare che qualunque grafo non orientato connesso $G = (V, E)$ soddisfa $|E| \geq |V| - 1$.

5.4-5 Verificare che in un grafo non orientato la relazione "raggiungibile da" è una relazione di equivalenza sui vertici del grafo. Quale delle tre proprietà di una relazione di equivalenza vale in generale per la relazione "raggiungibile da" sui vertici di un grafo orientato?

5.4-6 Qual è la versione non orientata del grafo orientato della figura 5.2 (a)? Qual è la versione orientata del grafo non orientato della figura 5.2 (b)?

* *5.4-7* Mostrare che un ipergrafo può essere rappresentato con un grafo bipartito se si assume che l'incidenza nell'ipergrafo corrisponda all'adiacenza nel grafo bipartito.

Figura 5.4 (a) *Un albero libero.* (b) *Una foresta.* (c) *Un grafo che contiene un ciclo e non è perciò né un albero né una foresta.*

(*Suggerimento*: Si supponga che uno degli insiemi dei vertici del grafo bipartito corrisponda ai vertici dell'ipergrafo e l'altro insieme di vertici corrisponda agli iperarchi.)

5.5 Alberi

Come per i grafi, ci sono molte nozioni di alberi, correlate, ma leggermente diverse. Questo paragrafo presenta definizioni e proprietà matematiche di molti tipi di alberi. I paragrafi 11.4 e 23.1 descrivono come gli alberi possano essere rappresentati nella memoria di un calcolatore.

5.5.1 Alberi liberi

Un ***albero libero***, come definito nel paragrafo 5.4, è un grafo non orientato, connesso e aciclico. Spesso si omette l'aggettivo "libero", quando si dice che un grafo è un albero. Quando un grafo non orientato è aciclico, ma può essere sconnesso, allora si dice che è una ***foresta***. Molti algoritmi che operano sugli alberi operano anche sulle foreste. La figura 5.4 (a) mostra un albero libero e la figura 5.4 (b) mostra una foresta. Il grafo nella figura 5.4 (c) non è né un albero né una foresta perché contiene un ciclo.

Il seguente teorema raccoglie molte importanti proprietà che riguardano gli alberi.

Teorema 5.2 (Proprietà degli alberi liberi)

Sia $G = (V, E)$ un grafo non orientato. Le seguenti affermazioni sono equivalenti.

1. G è un albero libero.
2. Due vertici qualsiasi in G sono connessi da un unico cammino semplice.
3. G è connesso, ma se un qualunque arco è rimosso da E, il grafo risultante è sconnesso.
4. G è connesso e $|E| = |V| - 1$.
5. G è aciclico e $|E| = |V| - 1$.
6. G è aciclico, ma se un qualsiasi arco è aggiunto ad E, il grafo risultante contiene un ciclo.

Figura 5.5 *Un passo della dimostrazione del Teorema 5.2: se* (1) *G è un albero libero, allora* (2) *qualsiasi coppia di vertici in G è connessa da un unico cammino semplice. Si assuma per assurdo che i vertici u e v siano connessi da due cammini semplici distinti p_1 e p_2. Questi cammini prima divergono dal vertice w e poi riconvergono nel vertice z. Il cammino p', concatenato con il cammino p'' considerato nel verso opposto, forma un ciclo da cui deriva la contraddizione.*

Dimostrazione. (1) ⇒ (2): Dato che un albero è connesso, ogni coppia di vertici in G è connessa da almeno un cammino semplice. Siano u e v vertici connessi da due distinti cammini semplici p_1 e p_2, come mostrato dalla figura 5.5. Sia w il primo vertice a partire dal quale i due cammini cominciano a divergere; cioè w è il primo vertice, sia di p_1 che di p_2, il cui successore in p_1 è un vertice x ed il cui successore in p_2 è un vertice y, dove $x \neq y$. Sia z il primo vertice a partire dal quale i due cammini convergono sullo stesso percorso: cioè z è il primo vertice dopo w, che compare sia in p_1 che in p_2. Sia p' il sottocammino di p_1 da w a z che attraversa x; e sia p'' il sottocammino di p_2 da w a z che attraversa y. I cammini p' e p'' non condividono vertici ad eccezione dei vertici iniziali e finali. Di conseguenza, il cammino ottenuto concatenando p' e p'' rovesciato sarebbe un ciclo, ma questa è una contraddizione. Quindi, se G è un albero ci può essere al più un cammino semplice tra due vertici.

(2) ⇒ (3): Se qualsiasi coppia di vertici in G è connessa da un unico cammino semplice, allora G è connesso. Sia (u, v) un qualunque arco in E. Questo arco rappresenta un cammino da u a v e così deve essere l'unico cammino da u a v. Se si rimuove (u, v) da G, allora non ci sono cammini da u a v e quindi la sua rimozione provoca la sconnessione di G.

(3) ⇒ (4): Per ipotesi, il grafo G è connesso e, per l'Esercizio 5.4-4, si ha che $|E| \geq |V| - 1$. Si proverà che $|E| \leq |V| - 1$ per induzione. Un grafo connesso con $n = 1$ o $n = 2$ vertici ha $n - 1$ archi. Si supponga che G abbia $n \geq 3$ vertici e che tutti i grafi che soddisfano la (3) con meno di n vertici soddisfino anche $|E| \leq |V| - 1$. Rimuovendo un arco arbitrario da G si separa il grafo in $k \geq 2$ componenti connesse (in realtà $k = 2$). Ogni componente soddisfa la (3), altrimenti G non avrebbe soddisfatto la (3). Quindi, per induzione, il numero di archi in tutte le componenti combinate è al più $|V| - k \leq |V| - 2$. Aggiungendo l'arco rimosso si ottiene $|E| \leq |V| - 1$.

(4) ⇒ (5): Si supponga che G sia connesso e che $|E| = |V| - 1$. Si deve mostrare che G è aciclico. Si supponga che G abbia un ciclo contenente k vertici $v_1, v_2, \ldots, v_k$. Sia $G_k = (V_k, E_k)$ il sottografo di G costituito dal ciclo. Si noti che $|V_k| = |E_k| = k$. Se $k < |V|$ deve esistere un vertice $V_{k+1} \in V - V_k$ che è adiacente a qualche vertice $v_i \in v_k$, perché G è connesso. Si definisca $G_{k+1} = (V_{k+1}, E_{k+1})$ il sottografo di G con $V_{k+1} = V_k \cup \{v_{k+1}\}$ e $E_{k+1} = E_k \cup \{(v_i, v_{k+1})\}$. Si noti che $|V_{k+1}| = |E_{k+1}| = k+1$. Se $k + 1 < n$, si può continuare nello stesso modo definendo G_{k+2}, e così via finché si ottiene $G_n = (V_n, E_n)$, dove $n = |V|$, $V_n = V$ e $|E_n| = |V_n| = |V|$. Poiché G_n è un sottografo di G, si ha $E_n \subseteq E$ e quindi $|E| \geq |V|$ che contraddice l'ipotesi che $|E| = |V| - 1$. Quindi G è aciclico.

(5) ⇒ (6): Si supponga che G sia aciclico e che $|E| = |V| - 1$. Sia k il numero di componenti connesse di G. Ogni componente connessa è un albero libero per definizione, e poiché la (1)

Figura 5.6 *Alberi radicati ed ordinati.* (a) *Un albero radicato di altezza 4. L'albero è disegnato nel modo standard: la radice (nodo 7) è in alto, i suoi figli (i nodi di profondità 1) sono appena sotto di essa, i loro figli (i nodi con profondità 2) sono appena sotto di loro e così via. Se l'albero è ordinato, l'ordinamento relativo tra i figli da sinistra a destra ha importanza, altrimenti non ne ha.* (b) *Un altro albero radicato. Come albero radicato è identico all'albero in (a), ma come albero ordinato è diverso, dato che i figli del nodo 3 si trovano in un ordine differente.*

implica la (5), la somma di tutti gli archi in tutte le componenti connesse di G è $|V| - k$. Di conseguenza si deve avere $k = 1$ e G è in realtà un albero. Poiché la (1) implica la (2), qualunque coppia di vertici è collegata da un unico cammino semplice. Perciò l'aggiunta di un qualunque arco a G crea un ciclo.

(6) ⇒ (1): Si supponga che G sia aciclico, ma che, aggiungendo un qualunque arco ad E, si crei un ciclo. Si deve mostrare che G è connesso. Siano u e v vertici arbitrari di G. Se u e v non sono già adiacenti, l'aggiunta dell'arco (u, v) crea un ciclo in cui tutti gli archi tranne (u, v) appartengono a G. Quindi vi è un cammino da u a v, e poiché u e v sono stati scelti arbitrariamente, G è connesso. ■

5.5.2 Alberi radicati ed alberi ordinati

Un ***albero radicato*** è un albero libero in cui uno dei vertici si distingue dagli altri. Il vertice diverso è chiamato la ***radice*** dell'albero. Spesso si fa riferimento a un vertice di un albero radicato come a un ***nodo***[2] dell'albero. La figura 5.6 (a) mostra un albero radicato su un insieme di 12 nodi con radice 7.

Si consideri un nodo x di un albero radicato T con radice r; qualunque nodo y sull'unico cammino da r a x è chiamato ***antenato*** di x e se y è un antenato di x, allora x è un ***discendente*** di y. (Ogni nodo è sia un antenato che un discendente di se stesso.) Se y è un antenato di x, e $x \neq y$, allora y è un ***antenato proprio*** di x e x è un ***discendente proprio*** di y. Il ***sottoalbero radicato in x*** è l'albero indotto dai discendenti di x, radicato in x. Per esempio, il sottoalbero radicato nel nodo 8 della figura 5.6 (a) contiene i nodi 8, 6, 5 e 9.

Se l'ultimo arco di un cammino dalla radice r di un albero T ad un nodo x è (y, x), allora y è ***il padre*** di x e x è ***il figlio*** di y; la radice è l'unico nodo in T che non ha padre. Se due nodi hanno lo stesso padre, si dicono fratelli. Un nodo senza figli si dice ***nodo esterno*** o ***foglia***. Un nodo non foglia è un ***nodo interno***.

[2] Il termine "nodo" è spesso usato nella letteratura sulla teoria dei grafi come sinonimo di "vertice". Noi useremo il termine "nodo" per indicare un vertice di un albero radicato.

Figura 5.7 *Alberi binari.* (a) *Un albero binario disegnato nel modo standard. Il figlio sinistro di un nodo è disegnato appena sotto al nodo, alla sua sinistra. Il figlio destro di un nodo è disegnato appena sotto al nodo alla sua destra.* (b) *Un albero binario diverso da quello in (a). In (a), il figlio sinistro del nodo 7 è 5 mentre il figlio destro è assente. In (b), il figlio sinistro del nodo 7 è assente mentre il figlio destro è 5. Come alberi ordinati, questi alberi rappresentano lo stesso albero, ma come alberi binari sono diversi.* (c) *L'albero binario in (a) rappresentato dai nodi interni di un albero binario pieno: un albero ordinato in cui ciascun nodo interno ha grado 2. Le foglie dell'albero sono disegnate con dei quadrati.*

Il numero di figli di un nodo x in un albero radicato T è chiamato il ***grado*** di x.[3] La lunghezza del cammino dalla radice r ad un nodo x è *la **profondità*** di x in T; la profondità più grande di un qualsiasi nodo in T è l'***altezza*** di T.

Un ***albero ordinato*** è un albero radicato in cui i figli di ciascun nodo sono ordinati. Cioè, se un nodo ha k figli, allora vi è un primo figlio, un secondo figlio, ..., ed un k-esimo figlio. I due alberi nella figura 5.6 sono diversi se sono considerati alberi ordinati, ma rappresentano lo stesso albero radicato.

5.5.3 Alberi binari e posizionali

Gli alberi binari sono descritti meglio in modo ricorsivo. Un ***albero binario*** T è una struttura definita su un insieme finito di nodi che:

- non contiene alcun nodo, oppure
- è composta da tre insiemi disgiunti di nodi: un nodo ***radice***, un albero binario chiamato il suo ***sottoalbero sinistro*** e un albero binario chiamato il suo ***sottoalbero destro***.

L'albero binario che non contiene alcun nodo è chiamato ***albero vuoto*** o ***albero nullo***, talvolta denotato con NIL. Se il sottoalbero sinistro non è vuoto la sua radice è chiamata ***figlio sinistro*** della radice dell'intero albero. Analogamente, la radice di un sottoalbero destro non nullo è il ***figlio destro*** della radice dell'intero albero. Se un sottoalbero è l'albero nullo NIL, si dice che il figlio è ***assente*** o mancante. La figura 5.7(a) mostra un albero binario.

Un albero binario non è semplicemente un albero ordinato in cui ogni nodo abbia grado al più 2. Per esempio, in un albero binario, se un nodo ha solo un figlio, la posizione del figlio – ***figlio sinistro*** o ***destro*** – è importante. In un albero ordinato, non si distingue se un figlio unico è destro o sinistro. La figura 5.7 (b) mostra un albero binario che differisce da quello

[3] Si noti che il grado di un nodo dipende dal fatto che T sia considerato un albero radicato o un albero libero. Il grado di un vertice di un albero libero è, come in qualsiasi grafo indiretto, il numero di vertici adiacenti. Tuttavia, in un albero radicato il grado è il numero di figli: il padre di un nodo non conta per determinarne il grado.

Figura 5.8 *Un albero binario completo di altezza 3 con 8 foglie e 7 nodi interni.*

nella figura 5.7(a) a causa della posizione di un nodo. Tuttavia, considerati come alberi ordinati i due alberi sono identici.

L'informazione sulla posizione in un albero binario può essere rappresentata dai nodi interni di un albero ordinato come mostrato nella figura 5.7 (c). L'idea è di sostituire ciascun figlio mancante nell'albero binario con un nodo che non ha figli; questi nodi foglia sono disegnati nella figura con dei quadrati. L'albero che si ottiene è un ***albero pienamente binario***: ogni nodo o è una foglia o ha esattamente grado due; non ci sono nodi di grado 1. Di conseguenza, l'ordine dei figli di un nodo mantiene l'informazione sulla posizione.

L'informazione sulla posizione che distingue gli alberi binari da quelli ordinati può essere estesa agli alberi con più di 2 figli per nodo. In un ***albero posizionale***, i figli di un nodo sono etichettati con interi positivi distinti. L'i-esimo figlio di un nodo è ***assente*** se nessun figlio è etichettato con l'intero i. Un albero ***k-ario*** è un albero posizionale in cui, per ogni nodo, tutti i figli con etichetta più grande di k sono assenti. Quindi un albero binario è un albero k-ario con $k = 2$.

Un ***albero k-ario completo*** è un albero k-ario in cui tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado k. La figura 5.8 mostra un albero binario completo di altezza 3. Quante foglie ha un albero k-ario completo con altezza h? La radice ha k figli a profondità 1, ciascuno dei quali ha k figli a profondità 2, etc. Di conseguenza, il numero di foglie a profondità h è k^h. In conclusione, l'altezza di un albero k-ario completo con n foglie è $\log_k n$. Il numero dei nodi interni di un albero k-ario completo di altezza h è

$$\begin{aligned} 1 + k + k^2 + \cdots + k^{h-1} &= \sum_{i=0}^{h-1} k^i \\ &= \frac{k^h - 1}{k - 1} \end{aligned}$$

per l'equazione (3.3). Quindi, un albero binario completo ha $2^h - 1$ nodi interni.

Esercizi

5.5-1 Disegnare tutti gli alberi liberi composti da tre vertici A, B e C. Disegnare tutti gli alberi radicati con nodi A, B e C, con A come radice. Disegnare tutti gli alberi ordinati con nodi A, B e C, con A come radice. Disegnare tutti gli alberi binari con nodi A, B e C, con A come radice.

5.5-2 Mostrare che, per $n \geq 7$, esiste un albero libero con n nodi tale che prendendo di volta in volta ciascuno degli n nodi come radice, si ottiene sempre un albero radicato diverso.

5.5-3 Sia $G = (V, E)$ un grafo aciclico orientato in cui vi è un vertice $v_0 \in V$ tale che esiste un unico cammino da v_0 ad ogni altro vertice $v \in V$. Provare che la versione non orientata di G forma un albero.

5.5-4 Mostrare per induzione che il numero dei nodi di grado 2 in qualunque albero binario è uguale al numero di foglie meno 1.

5.5-5 Mostrare per induzione che un albero binario con n nodi ha un'altezza di almeno $\lfloor \lg n \rfloor$.

* **5.5-6** La ***lunghezza dei cammini interni*** di un albero pienamente binario è la somma, presa su tutti i nodi interni dell'albero, delle profondità di ciascun nodo. Analogamente, la ***lunghezza dei cammini esterni*** è la somma, presa su tutte le foglie dell'albero, delle profondità di ciascuna foglia. Considerare un albero pienamente binario con n nodi interni, lunghezza dei cammini interni i e lunghezza dei cammini esterni e. Provare che $e = i + 2n$.

* **5.5-7** Si associ un "peso" $w(x) = 2^{-d}$ a ciascuna foglia x di profondità d di un albero binario T. Mostrare che $\sum_x w(x) \leq 1$, dove la somma è presa su tutte le foglie x di T. (Questa è nota come la ***disuguaglianza di Kraft***.)

* **5.5-8** Mostrare che ogni albero binario con L foglie contiene un sottoalbero che ha un numero di foglie compreso tra $L/3$ e $2L/3$, estremi inclusi.

Problemi

5-1 *Colorazione di un grafo*

Una ***k-colorazione*** di un grafo non orientato $G = (V, E)$ è una funzione $c : V \rightarrow \{0, 1, \ldots, k-1\}$ tale che $c(u) \neq c(v)$ per ogni arco $(u, v) \in E$. In altre parole, i numeri $0, 1, \ldots, k-1$ rappresentano k colori e vertici adiacenti devono avere colori diversi.

a. Mostrare che qualsiasi albero è 2-colorabile.

b. Mostrare che le seguenti affermazioni sono equivalenti:

1. G è bipartito.
2. G è 2-colorabile.
3. G non ha cicli di lunghezza dispari.

c. Sia d il massimo grado di un qualsiasi vertice di un grafo G. Provare che G può essere colorato con $d + 1$ colori.

d. Mostrare che se G ha $O(|V|)$ archi, allora G può essere colorato con $O(\sqrt{|V|})$ colori.

5-2 *Grafi "amichevoli"*

Riformulare ciascuna delle seguenti affermazioni come un teorema sui grafi non orientati e quindi darne una dimostrazione. Si assuma che la relazione di amicizia sia simmetrica ma non riflessiva.

a. In un generico gruppo di $n \geq 2$ persone, vi sono due persone con lo stesso numero di amici nel gruppo.

b. Ogni gruppo di sei persone contiene o tre amici reciproci o tre non conoscenti reciproci.

c. Qualsiasi gruppo di persone può essere partizionato in due sottogruppi tali che almeno la metà degli amici di ciascuna persona appartiene al sottogruppo a cui quella persona *non* appartiene.

d. Se in un gruppo ciascuno è amico di almeno la metà delle persone del gruppo, allora il gruppo può essere fatto sedere ad un tavolo in modo che ognuno sia seduto tra due amici.

5-3 *Bisezione di alberi*

Molti algoritmi divide-et-impera che operano sui grafi richiedono che il grafo sia bisecato in due sottografi di dimensione simile rimuovendo un piccolo numero di archi. Questo problema studia la bisezione di alberi.

a. Mostrare che rimuovendo un singolo arco, si possono partizionare i vertici di un albero binario con n vertici in due insiemi A e B tali che $|A| \leq 3n/4$ e $|B| \leq 3n/4$.

b. Mostrare che la costante $3/4$ del quesito (a) è ottima nel caso peggiore fornendo un esempio di un semplice albero la cui partizione più equamente bilanciata, ottenuta togliendo un singolo arco, fornisca $|A| = 3n/4$.

c. Mostrare che rimuovendo al più $O(\lg n)$ archi, si possono partizionare i vertici di un qualsiasi albero con n vertici in due insiemi A e B tali che $|A| = \lfloor n/2 \rfloor$ e $|B| = \lceil n/2 \rceil$.

Note al capitolo

G. Boole fu uno dei primi studiosi della logica simbolica e introdusse molta della notazione insiemistica di base in un libro pubblicato nel 1854. La teoria insiemistica moderna prese avvio con G. Cantor nel periodo 1874-1895. Cantor si interessò principalmente agli insiemi di cardinalità infinita. Il termine "funzione" è attribuito a G. W. Leibnitz, che lo usò per riferirsi a diversi tipi di formule matematiche; la sua definizione limitata è stata generalizzata molte volte. La teoria dei grafi ha origine nel 1736, quando L. Eulero dimostrò che era impossibile attraversare i sette ponti della città di Königsberg passandovi sopra esattamente una volta e tornando al punto di partenza.

Un utile compendio di molte definizioni e risultati sulla teoria dei grafi è il libro di Harary [94].

Calcolo combinatorio e delle probabilità 6

Questo capitolo offre una panoramica della teoria del calcolo combinatorio elementare e del calcolo delle probabilità. Se il lettore ha già una buona preparazione su questi argomenti, può sfogliare l'inizio del capitolo e concentrarsi sui paragrafi successivi; la maggior parte dei capitoli successivi non richiede nozioni di probabilità, tranne qualcuno in cui sono invece essenziali.

Il paragrafo 6.1 presenta i risultati elementari della teoria del calcolo combinatorio, incluse formule standard per contare le permutazioni e le combinazioni. Gli assiomi sulla probabilità e i concetti di base che riguardano le distribuzioni di probabilità sono presentati nel paragrafo 6.2. Le variabili casuali sono introdotte nel paragrafo 6.3, insieme con le proprietà di speranza e varianza. Il paragrafo 6.4 considera le distribuzioni geometrica e binomiale che si incontrano studiando le prove di Bernoulli. Lo studio della distribuzione binomiale è continuato nel paragrafo 6.5, con una discussione avanzata delle "code" di tale distribuzione. Infine, il paragrafo 6.6 illustra l'analisi probabilistica attraverso tre esempi: il paradosso del compleanno, il lancio casuale di palline in contenitori e le sequenze vincenti.

6.1 Calcolo combinatorio

La teoria del calcolo combinatorio cerca di rispondere alla domanda "Quanti?" senza di fatto contare. Per esempio, si potrebbe chiedere: "Quanti sono i numeri distinti di n bit?" oppure "Quanti sono gli ordinamenti distinti di n elementi?" In questo paragrafo, si passeranno in rassegna gli elementi della teoria del calcolo combinatorio. Poiché spesso si assume una conoscenza di base degli insiemi, si consiglia al lettore di rivedere il materiale del paragrafo 5.1.

Regole di somma e prodotto

Un insieme di elementi che si desidera contare può talvolta essere espresso come unione di insiemi disgiunti o prodotto cartesiano di insiemi.

La ***regola della somma*** dice che il numero di modi per scegliere un elemento da uno di due insiemi *disgiunti* è la somma delle cardinalità degli insiemi. Cioè, se A e B sono due insiemi finiti senza elementi in comune, allora $|A \cup B| = |A| + |B|$, che segue dall'equazione (5.3). Per esempio, il numero di targa di una macchina ha in ogni posizione una lettera o una cifra. Il numero di possibilità per ogni posizione è allora $26 + 10 = 36$, infatti ci sono 26 possibili scelte se è una lettera e 10 scelte se è una cifra.

La ***regola del prodotto*** dice che il numero di modi per scegliere una coppia ordinata è il numero di modi per scegliere il primo elemento moltiplicato il numero di modi per scegliere il secondo. Cioè, se A e B sono due insiemi finiti, allora $|A \times B| = |A| \cdot |B|$, che è semplicemente l'equazione (5.4). Per esempio se una gelateria offre 28 gusti di gelato e 4 tipi di cialde, il numero di coppe possibili con una pallina di gelato e una cialda è $28 \cdot 4 = 112$.

Stringhe

Una ***stringa*** su un insieme finito S è una sequenza di elementi di S. Per esempio le stringhe binarie di lunghezza 3 sono 8:

000, 001, 010, 011, 100, 101, 110, 111.

Talvolta una stringa di lunghezza k sarà chiamata ***k-stringa***. Una ***sottostringa*** s' di una stringa s è una sequenza ordinata di elementi consecutivi di s. Una ***k-sottostringa*** di una stringa è una sottostringa di lunghezza k. Per esempio, 010 è una 3-sottostringa di 01101001 (la 3-sottostringa che comincia in posizione 4), ma 111 non è una sottostringa di 01101001.

Una k-stringa su un insieme S può essere vista come un elemento del prodotto cartesiano S^k di k-ple; quindi le stringhe di lunghezza k sono $|S|^k$. Per esempio, il numero di k-stringhe binarie è 2^k. Intuitivamente, per costruire una k-stringa su un n-insieme, si hanno n modi per scegliere il primo elemento; per ognuna di queste scelte, si hanno n modi per scegliere il secondo elemento; e così via fino a k volte. Questa costruzione porta al prodotto $n \cdot n \cdot \ldots \cdot n = n^k$ come numero delle k-stringhe.

Permutazioni

Una ***permutazione*** di un insieme finito S è una sequenza ordinata di tutti gli elementi di S, in cui ogni elemento compare esattamente una volta. Per esempio se $S = \{a, b, c\}$, ci sono 6 permutazioni di S:

abc, acb, bac, bca, cab, cba.

Le permutazioni di un insieme di n elementi sono $n!$, dato che il primo elemento della sequenza può essere scelto in n modi, il secondo in $n-1$ modi, il terzo in $n-2$ modi e così via.

Una ***k-permutazione*** di S è una sequenza ordinata di k elementi di S in cui gli elementi non compaiono più di una volta. (Quindi, una permutazione ordinaria è esattamente una n-permutazione di un n-insieme.) Le dodici 2- permutazioni dell'insieme $\{a, b, c, d\}$ sono:

ab, ac, ad, ba, bc, bd, ca, cb, cd, da, db, dc.

Il numero delle k permutazioni di un n-insieme è

$$n(n-1)(n-2)\cdots(n-k+1) = \frac{n!}{(n-k)!}, \tag{6.1}$$

poiché vi sono n modi di scegliere il primo elemento, $n-1$ modi di scegliere il secondo e così via finché vengono selezionati k elementi, e il k-esimo viene scelto tra $n-k+1$ elementi.

Combinazioni

Una ***k-combinazione*** di un n-insieme S è semplicemente un k-sottoinsieme di S. Vi sono sei 2-combinazioni del 4-insieme $\{a, b, c, d\}$:

ab, ac, ad, bc, bd, cd.

(Si è usata l'abbreviazione *ab* per denotare il 2-insieme $\{a, b\}$.) Si può costruire una k-combinazione di un n-insieme scegliendo in esso k elementi distinti (diversi).

Il numero di k-combinazioni di un n-insieme può essere espresso in termini del numero di k-permutazioni di un n-insieme. Per ogni k-combinazione, vi sono esattamente $k!$ permutazioni dei suoi elementi, ognuno dei quali è una diversa k-permutazione dell'n-insieme. Quindi, il numero di k-combinazioni di un n-insieme è il numero di k-permutazioni diviso $k!$: dall'equazione (6.1), questa quantità è

$$\frac{n!}{k!(n-k)!}. \tag{6.2}$$

Per $k = 0$, questa formula dice che il numero di modi per scegliere 0 elementi da un n-insieme è 1, infatti $0! = 1$.

Coefficienti binomiali

Si usa la notazione $\binom{n}{k}$ (si legga "n su k") per denotare il numero di k-combinazioni di un n-insieme. Dall'equazione (6.2), si ha

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{6.3}$$

Questa formula è simmetrica per k ed $n-k$, cioè

$$\binom{n}{k} = \binom{n}{n-k}. \tag{6.4}$$

Questi numeri sono conosciuti anche come ***coefficienti binomiali***, a causa del fatto che compaiono nello ***sviluppo binomiale***

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}. \tag{6.5}$$

Un caso speciale di sviluppo binomiale si verifica quando $x = y = 1$:

$$2^n = \sum_{k=0}^{n} \binom{n}{k}. \tag{6.6}$$

Questa formula corrisponde a contare le 2^n n-stringhe binarie tramite il numero di 1 che contengono: vi sono $\binom{n}{k}$ n-stringhe binarie che contengono esattamente k volte un 1, perché vi sono $\binom{n}{k}$ modi di scegliere k posizioni in cui mettere gli 1 tra le n posizioni possibili.

Sono note molte identità che riguardano i coefficienti binomiali. Gli esercizi alla fine di questo paragrafo forniscono l'opportunità di dimostrarne alcune.

Limiti binomiali

Si ha talvolta la necessità di limitare la grandezza di un coefficiente binomiale. Per $1 \le k \le n$, si ha il limite inferiore

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}\\ &= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right)\cdots\left(\frac{n-k+1}{1}\right)\\ &\geq \left(\frac{n}{k}\right)^k .\end{aligned} \tag{6.7}$$

Sfruttando la disuguaglianza $k! \geq (k/e)^k$ che deriva dalla formula di Stirling (2.12) si ottengono i limiti superiori

$$\begin{aligned}\binom{n}{k} &= \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}\\ &\leq \frac{n^k}{k!} &(6.8)\\ &\leq \left(\frac{en}{k}\right)^k . &(6.9)\end{aligned}$$

Per ogni $0 \leq k \leq n$, si può usare l'induzione (si veda l'Esercizio 6.1-12) per dimostrare il limite

$$\binom{n}{k} \leq \frac{n^n}{k^k(n-k)^{n-k}} , \tag{6.10}$$

dove, per comodità, si assume che $0^0 = 1$. Per $k = \lambda n$, dove $0 \leq \lambda \leq 1$, questo limite può essere riscritto come

$$\begin{aligned}\binom{n}{\lambda n} &\leq \frac{n^n}{(\lambda n)^{\lambda n}((1-\lambda)n)^{(1-\lambda)n}}\\ &= \left(\left(\frac{1}{\lambda}\right)^{\lambda}\left(\frac{1}{1-\lambda}\right)^{1-\lambda}\right)^n &(6.11)\\ &= 2^{nH(\lambda)} , &(6.12)\end{aligned}$$

dove

$$H(\lambda) = -\lambda \lg \lambda - (1-\lambda)\lg(1-\lambda) \tag{6.13}$$

è la ***funzione di entropia*** (***binaria***) e dove, per comodità, si assume che $0 \lg 0 = 0$, in modo che $H(0) = H(1) = 0$.

Esercizi

6.1-1 Quante k-sottostringhe ha una n-stringa? (Si considerino diverse le k-sottostringhe identiche che si trovano in posizioni diverse.) Quante sottostringhe ha in totale una n-stringa?

6.1-2 Una ***funzione booleana*** con n input e m output è una funzione da $\{\text{TRUE}, \text{FALSE}\}^n$ a $\{\text{TRUE}, \text{FALSE}\}^m$. Quante sono le funzioni booleane con n input e 1 output? Quante quelle con n input e m output?

6.1-3 In quanti modi n professori si possono sedere attorno a un tavolo tondo? Si considerino identiche due situazioni in cui una può essere ruotata per formare l'altra.

6.1-4 In quanti modi si possono scegliere tre diversi numeri dall'insieme $\{1, 2, \ldots, 100\}$ così che la loro somma sia pari?

6.1-5 Provare l'identità

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{6.14}$$

per $0 < k \leq n$.

6.1-6 Provare l'identità

$$\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$$

per $0 \leq k < n$.

6.1-7 Per scegliere k oggetti su n, si può contraddistinguere uno degli oggetti e considerare se quell'oggetto viene scelto. Usare quest'approccio per provare che

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} .$$

6.1-8 Usando il risultato dell'Esercizio 6.1-7, costruire una tabella per $n = 0, 1, \ldots, 6$ e $0 \leq k \leq n$ dei coefficienti binomiali $\binom{n}{k}$ con $\binom{0}{0}$ in alto, $\binom{1}{0}$ e $\binom{1}{1}$ sulla linea successiva, e così via. Tale tabella di coefficienti binomiali è chiamata ***triangolo di Pascal***.

6.1-9 Provare che

$$\sum_{i=1}^{n} i = \binom{n+1}{2} .$$

6.1-10 Mostrare che per qualunque $n \geq 0$ e $0 \leq k \leq n$, il massimo valore di $\binom{n}{k}$ è raggiunto quando $k = \lfloor n/2 \rfloor$ o $k = \lceil n/2 \rceil$.

* *6.1-11* Dedurre che per qualunque $n \geq 0$, $j \geq 0$, $k \geq 0$ con $j + k \leq n$,

$$\binom{n}{j+k} \leq \binom{n}{j}\binom{n-j}{k} . \tag{6.15}$$

Fornire sia una prova algebrica che una motivazione basata su un metodo per scegliere $j + k$ elementi su n. Fornire un esempio in cui l'uguaglianza non vale.

* *6.1-12* Usare l'induzione su $k \leq n/2$ per provare la disuguaglianza (6.10) e usare l'equazione (6.4) per estenderla a tutti i $k \leq n$.

* *6.1-13* Usare l'approssimazione di Stirling per provare che

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}}(1 + O(1/n)) . \tag{6.16}$$

* *6.1-14* Differenziando la funzione di entropia $H(\lambda)$, mostrare che essa raggiunge il suo valore massimo per $\lambda = 1/2$. Quanto vale $H(1/2)$?

6.2 Calcolo delle probabilità

Il calcolo delle probabilità è uno strumento essenziale per la progettazione e l'analisi degli algoritmi probabilistici e di quelli randomizzati. Questo paragrafo presenta una panoramica delle basi della teoria della probabilità.

Si definisce la probabilità in termini di uno ***spazio campione*** S, che è un insieme i cui elementi sono chiamati ***eventi elementari***. Ciascun evento elementare può essere considerato come uno dei possibili esiti di una prova. Per la prova del lancio di due monete distinguibili, si può assumere uno spazio campione composto dall'insieme di tutte le possibili 2-stringhe su {T, C} (T sta per Testa mentre C sta per Croce):

$S = \{$TT, TC, CT, CC$\}$.

Un ***evento***[1] è un sottoinsieme dello spazio campione S. Per esempio, nell'esperimento del lancio di due monete, ottenere una testa ed una croce è l'evento {TC, CT}. L'evento S è chiamato ***evento certo*** e l'evento $\varnothing$ è chiamato l'***evento nullo***. Si dice che due eventi A e B sono ***mutuamente esclusivi*** se $A \cap B = \varnothing$. Talvolta si tratterà un evento elementare $s \in S$ come l'evento $\{s\}$. Per definizione, tutti gli eventi elementari sono mutuamente esclusivi.

Assiomi della probabilità

Una ***distribuzione di probabilità*** Pr{ } su uno spazio campione S è una corrispondenza tra gli eventi di S ed i numeri reali tale che siano soddisfatti i seguenti ***assiomi della probabilità***:

1. $\Pr\{A\} \geq 0$ per ogni evento A.
2. $\Pr\{S\} = 1$.
3. $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ per qualsiasi coppia di eventi mutuamente esclusivi A e B. Più in generale, per qualsiasi sequenza (finita o numerabile) di eventi $A_1, A_2, \ldots$ che sono due a due mutuamente esclusivi,

$$\Pr\left\{\bigcup_i A_i\right\} = \sum_i \Pr\{A_i\} .$$

[1] In generale, per una distribuzione di probabilità, vi possono essere alcuni sottoinsiemi dello spazio campione S che non sono considerati eventi. Questa situazione si verifica generalmente quando lo spazio campione è un insieme infinito non numerabile. Il requisito principale è che l'insieme degli eventi di uno spazio campione sia chiuso rispetto alle operazioni di calcolo del complemento di un evento, dell'unione di un numero finito o numerabile di eventi e dell'intersezione di un numero finito o numerabile di eventi. La maggior parte delle distribuzioni di probabilità che si vedranno sono su spazi campione finiti o numerabili, considerando, in generale, come possibili eventi tutti i sottoinsiemi dello spazio campione. La distribuzione di probabilità continua uniforme, che sarà presentata tra breve, rappresenta un'eccezione.

Il valore di $\Pr\{A\}$ è chiamato la ***probabilità*** di A. Si noti che l'assioma 2 è un requisito di normalizzazione: in realtà non vi è niente che obblighi a scegliere 1 come la probabilità dell'evento certo, eccetto il fatto che è semplice e comodo.

Da questi assiomi e dalla teoria di base degli insiemi (si veda il paragrafo 5.1) seguono immediatamente diversi risultati. L'evento nullo $\varnothing$ ha probabilità $\Pr\{\varnothing\} = 0$. Se $A \subseteq B$, allora $\Pr\{A\} \leq \Pr\{B\}$. Usando $\overline{A}$ per denotare l'evento $S - A$ (***complemento*** di A), si ha che $\Pr\{\overline{A}\} = 1 - \Pr\{A\}$. Per ogni coppia di eventi A e B,

$$\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \tag{6.17}$$

$$\leq \Pr\{A\} + \Pr\{B\} . \tag{6.18}$$

Nell'esempio del lancio delle monete di cui sopra, si può supporre che i quattro eventi elementari abbiano probabilità 1/4. Quindi la probabilità di ottenere almeno una testa è

$$\begin{aligned}\Pr\{\text{TT, TC, CT}\} &= \Pr\{\text{TT}\} + \Pr\{\text{TC}\} + \Pr\{\text{CT}\} \\ &= 3/4\end{aligned}$$

Alternativamente, dato che la probabilità di ottenere un risultato senza nemmeno una testa è $\Pr\{\text{CC}\} = 1/4$, la probabilità di ottenere almeno una testa è $1 - 1/4 = 3/4$.

Distribuzione discreta di probabilità

Una distribuzione di probabilità è ***discreta*** se è definita su uno spazio campione finito o numerabile. Sia S uno spazio campione, allora per ogni evento A,

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\} ,$$

dato che gli eventi elementari in A sono mutuamente esclusivi. Se S è finito ed ogni evento elementare $s \in S$ ha probabilità

$$\Pr\{s\} = 1/|S| ,$$

si ha una ***distribuzione uniforme di probabilità*** su S. In tal caso l'esperimento è spesso descritto come "estrarre in modo casuale un elemento da S".

Come esempio si può considerare il processo del lancio di una ***moneta perfetta***, cioè una moneta per la quale la probabilità di ottenere testa è uguale a quella di ottenere croce, cioè 1/2. Se si lancia la moneta n volte, si ha una distribuzione uniforme di probabilità definita sullo spazio campione $S = \{\text{T, C}\}^n$, un insieme di cardinalità 2^n. Ciascun evento elementare di S può essere rappresentato come una stringa di lunghezza n su {T, C} e ciascuno di essi può verificarsi con probabilità $1/2^n$. L'evento

$A = \{$risultano esattamente k teste ed $n - k$ croci$\}$

è un sottoinsieme di S di cardinalità $|A| = \binom{n}{k}$, dato che ci sono $\binom{n}{k}$ stringhe di lunghezza n su {T, C} che contengono esattamente k T. La probabillità dell'evento A è quindi

$\Pr\{A\} = \binom{n}{k}/2^n$.

Distribuzione continua uniforme di probabilità

La distribuzione ***continua*** uniforme di probabilità è un esempio di distribuzione di probabilità in cui non tutti i sottoinsiemi dello spazio campione sono considerati eventi. La

distribuzione continua uniforme di probabilità è definita su un intervallo chiuso $[a, b]$ di numeri reali, dove $a < b$. Il concetto intuitivo è che si vuole che ciascun punto dell'intervallo sia "equamente probabile". Tuttavia, il numero di punti è non numerabile e quindi se a tutti i punti è data la stessa probabilità finita e positiva, non si possono soddisfare contemporaneamente gli assiomi 2 e 3. Per questa ragione, si associa una probabilità solo ad *alcuni* sottoinsiemi di S in modo tale che gli assiomi siano soddisfatti per tali eventi.

Per qualsiasi intervallo chiuso $[c, d]$, dove $a \le c \le d \le b$, la ***distribuzione continua uniforme di probabilità*** definisce la probabilità dell'evento $[c, d]$ come

$$\Pr\{[c,d]\} = \frac{d-c}{b-a} .$$

Si noti che per ogni punto $x = [x, x]$, la probabilità di x è 0. Rimuovendo i punti estremi di un intervallo $[c, d]$, si ottiene l'intervallo aperto (c, d). Dato che $[c, d] = [c, c] \cup (c, d) \cup [d, d]$, per l'assioma 3 si ha $\Pr\{[c, d]\} = \Pr\{(c, d)\}$. In generale, l'insieme degli eventi di una distribuzione continua uniforme di probabilità è un qualsiasi intervallo $[a, b]$ di numeri reali che può essere ottenuto dall'unione di un numero finito o numerabile di intervalli aperti o chiusi.

Probabilità condizionata ed indipendenza

Talvolta si ha a priori una conoscenza parziale del risultato di un esperimento. Per esempio, si supponga che un amico abbia lanciato due monete perfette e che abbia detto che almeno una di queste monete mostrava testa: qual è la probabilità che entrambe le monete mostrassero testa? L'informazione fornita elimina la possibilità che siano uscite due croci; inoltre i tre eventi elementari rimanenti sono equamente probabili per cui si possono presentare con una probabilità di 1/3. Di conseguenza soltanto uno di questi eventi mostra due teste e la risposta alla domanda è 1/3.

La probabilità condizionata formalizza la nozione di avere a priori una conoscenza parziale del risultato di un esperimento. La ***probabilità condizionata*** di un evento A, dato che si è verificato un altro evento B, è definita come

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}} \tag{6.19}$$

tutte le volte che $\Pr\{B\} \neq 0$. (Leggere "$\Pr\{A|B\}$" come "la probabilità di A dato B"). Intuitivamente, dato che si è verificato l'evento B, l'evento che anche A si verifichi è $A \cap B$. Cioè $A \cap B$ è l'insieme dei risultati in cui si verificano sia A che B. Poiché il risultato è uno degli eventi elementari di B, si normalizzano le probabilità di tutti gli eventi elementari di B dividendoli per $\Pr\{B\}$, in modo che la loro somma sia 1. La probabilità condizionata di A dato B è perciò il rapporto tra la probabilità dell'evento $A \cap B$ e la probabilità dell'evento B. Nell'esempio precedente, A è l'evento che entrambe le monete mostrino testa e B è l'evento che almeno una moneta mostri testa. Quindi, $\Pr\{A|B\} = (1/4)/(3/4) = 1/3$.

Due eventi sono ***indipendenti*** se

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\} ,$$

che è equivalente, se $\Pr\{B\} \neq 0$, alla condizione che sia

$$\Pr\{A \mid B\} = \Pr\{A\} .$$

Per esempio, si supponga di lanciare due monete perfette e che i risultati siano indipendenti, allora la probabilità di ottenere due teste è $(1/2)(1/2) = 1/4$. Si supponga adesso che un evento

corrisponda al fatto che la prima moneta mostri testa e l'altro che le due facce mostrate siano diverse. Ognuno di questi eventi accade con probabilità 1/2 e la probabilità che entrambi gli eventi accadano è 1/4; quindi, in accordo con la definizione di indipendenza, gli eventi sono indipendenti, anche se si potrebbe pensare che entrambi gli eventi dipendano dalla prima moneta. Infine, si supponga che le monete siano saldate insieme in modo che cadano di testa o di croce contemporaneamente e che le due possibilità siano equamente probabili. Allora la probabilità che ogni moneta mostri testa è 1/2, ma la probabilità che mostrino testa entrambe è $1/2 \neq (1/2)(1/2)$. Di conseguenza, l'evento che una venga testa e l'evento che mostrino testa entrambe non sono indipendenti.

Gli eventi $A_1, A_2, \ldots, A_n$ di un insieme si dicono ***indipendenti a coppie*** se

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\}$$

per ciascun $1 \le i \le j \le n$. Si dice che essi sono ***(mutuamente) indipendenti*** se ogni k-sottoinsieme $A_{i_1}, A_{i_2}, \ldots A_{i_k}$ dell'insieme, dove $2 \le k \le n$ e $1 \le i_1 < i_2 < \ldots < i_k \le n$, soddisfa

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\}\cdots\Pr\{A_{i_k}\} .$$

Per esempio, si supponga di lanciare due monete perfette. Sia A_1 l'evento per cui la prima moneta dà testa, A_2 l'evento per cui la seconda dà testa e A_3 l'evento per cui i due risultati siano differenti. Si ha

$$\begin{aligned}
\Pr\{A_1\} &= 1/2 , \\
\Pr\{A_2\} &= 1/2 , \\
\Pr\{A_3\} &= 1/2 , \\
\Pr\{A_1 \cap A_2\} &= 1/4 , \\
\Pr\{A_1 \cap A_3\} &= 1/4 , \\
\Pr\{A_2 \cap A_3\} &= 1/4 , \\
\Pr\{A_1 \cap A_2 \cap A_3\} &= 0 .
\end{aligned}$$

Poiché, per $1 \le i < j \le 3$, si ha $\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} = 1/4$, i tre eventi sono indipendenti a coppie; tuttavia, gli eventi non sono mutuamente indipendenti perché $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$, mentre $\Pr\{A_1\}\Pr\{A_2\}\Pr\{A_3\} = 1/8 \neq 0$.

Il teorema di Bayes

Dalla definizione (6.19) della probabilità condizionata, segue che per due eventi A e B ognuno con probabilità non nulla,

$$\begin{aligned}
\Pr\{A \cap B\} &= \Pr\{B\}\Pr\{A \mid B\} \\
&= \Pr\{A\}\Pr\{B \mid A\} .
\end{aligned} \tag{6.20}$$

Risolvendo rispetto a $\Pr\{A|B\}$, si ottiene

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{B\}} , \tag{6.21}$$

formula conosciuta come ***teorema di Bayes***. Il denominatore è una costante di normalizzazione che può essere espressa anche nel modo seguente. Dato che $B = (B \cap A) \cup (B \cap \bar{A})$ e che gli eventi $B \cap A$ e $B \cap \bar{A}$ sono mutuamente esclusivi, si ha

$$\begin{aligned}\Pr\{B\} &= \Pr\{B \cap A\} + \Pr\{B \cap \overline{A}\} \\ &= \Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\} .\end{aligned}$$

Sostituendo questa formula nella (6.21), si ottiene una forma equivalente del teorema di Bayes:

$$\Pr\{A \mid B\} = \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\{\overline{A}\}\Pr\{B \mid \overline{A}\}} .$$

Il teorema di Bayes può semplificare il calcolo delle probabilità condizionate. Per esempio, si supponga di avere una moneta perfetta ed una moneta truccata per cui si ottiene sempre testa. Si può eseguire un esperimento che consiste di tre eventi indipendenti: una delle due monete è scelta a caso, è lanciata una volta e dopo è lanciata di nuovo; si supponga che esca testa entrambe le volte, qual è la probabilità che la moneta sia truccata?

Si risolve il problema usando il teorema di Bayes. Sia A l'evento che corrisponde alla scelta della moneta truccata, sia B l'evento di ottenere testa entrambe le volte. Si desidera determinare $\Pr\{A|B\}$. Si ha $\Pr\{A\} = 1/2$, $\Pr\{B|A\} = 1$, $\Pr\{\overline{A}\} = 1/2$ e $\Pr\{B|\overline{A}\} = 1/4$, quindi,

$$\begin{aligned}\Pr\{A \mid B\} &= \frac{(1/2)\cdot 1}{(1/2)\cdot 1 + (1/2)\cdot(1/4)} \\ &= 4/5 .\end{aligned}$$

Esercizi

6.2-1 Dimostrare la ***disuguaglianza di Boole***: per qualsiasi sequenza finita o numerabile di eventi $A_1, A_2, \ldots$,

$$\Pr\{A_1 \cup A_2 \cup \cdots\} \le \Pr\{A_1\} + \Pr\{A_2\} + \cdots . \qquad (6.22)$$

6.2-2 Il professor Rosencrantz lancia una moneta perfetta. Il professor Guildenstern lancia due monete perfette. Qual è la probabilità che il professor Rosencrantz ottenga più teste del professor Guildenstern?

6.2-3 Un mazzo di 10 carte, ognuna con un numero distinto da 1 a 10, viene mescolato. Tre carte sono rimosse dal mazzo una alla volta. Qual è la probabilità che le tre carte siano scelte ordinate in modo crescente?

* *6.2-4* Sia data una moneta truccata per cui si ottiene testa con probabilità (sconosciuta) p, dove $0 < p < 1$. Mostrare come un "lancio di moneta" equo possa essere simulato osservando lanci multipli della moneta truccata. (*Suggerimento*: lanciare la moneta due volte e poi determinare il risultato del lancio equo simulato oppure ripetere l'esperimento). Dimostrare che la risposta è corretta.

* *6.2-5* Descrivere una procedura che accetti come input due interi a e b tali che $0 < a < b$ e che, usando lanci di moneta perfetta, produca come risultato testa con probabilità a/b e croce con probabilità $(b - a)/b$. Dare un limite al numero atteso di lanci della moneta, che dovrebbe essere polinomiale in $\lg b$.

6.2-6 Provare che

$$\Pr\{A \mid B\} + \Pr\{\overline{A} \mid B\} = 1 .$$

6.2-7 Dimostrare che, per un qualunque insieme di eventi $A_1, A_2, \ldots, A_n$,

$$\Pr\{A_1 \cap A_2 \cap \cdots \cap A_n\} = \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_n \mid A_1 \cap A_2 \cap \cdots \cap A_{n-1}\} .$$

* *6.2-8* Mostrare come costruire un insieme di n eventi indipendenti a coppie, ma tale che per qualsiasi suo sottoinsieme di $k > 2$ elementi, questi *non* siano mutuamente indipendenti.

* *6.2-9* Due eventi A e B sono ***indipendenti in modo condizionato***, dato C, se

$$\Pr\{A \cap B \mid C\} = \Pr\{A \mid C\} \cdot \Pr\{B \mid C\} .$$

Fornire un esempio semplice ma non banale di due eventi che non sono indipendenti ma che sono indipendenti in modo condizionato, dato un terzo evento.

* *6.2-10* Si supponga di essere un concorrente di un gioco televisivo e che un premio sia nascosto dietro una fra tre porte. Si vince il premio se si sceglie la porta giusta. Dopo che una certa porta è stata scelta ma prima che venga aperta, il conduttore del programma apre una delle altre due porte, rivelando che dietro non c'è il premio; quindi chiede se si vuole cambiare la porta scelta. Se si cambia porta come varierà la probabilità di vincere?

* *6.2-11* Il direttore di una prigione ha scelto a caso uno fra tre prigionieri a cui restituire la libertà mentre gli altri due saranno giustiziati. Il secondino sa quale sarà liberato ma gli è proibito dare a qualsiasi prigioniero informazioni sulla sua sorte. Si indichino i prigionieri con X, Y e Z. Il prigioniero X chiede privatamente al secondino quale tra Y o Z sarà giustiziato: poiché già si conosce che almeno uno di loro deve morire, il secondino non gli rivelerà alcuna informazione sulla sua sorte. Il secondino dice a X che Y sarà giustiziato. Il prigioniero X adesso si sente più sollevato, poiché si immagina che o lui o il prigioniero Z sarà liberato, ciò significa che la sua probabilità di salvarsi è adesso 1/2. Ha ragione o le sue possibilità rimangono 1/3? Motivare la risposta.

6.3 Variabili casuali discrete

Una ***variabile casuale (discreta)*** X è una funzione che va da uno spazio campione S finito o numerabile, ai numeri reali. Essa associa un numero reale ad ogni possibile risultato di un esperimento: ciò permette di lavorare sulla distribuzione di probabilità indotta sul risultante insieme di numeri reali. Le variabili casuali possono essere definite anche su spazi campione non numerabili, ma ciò richiede dettagli tecnici che non risultano necessari per i nostri obiettivi. Per cui, si assumerà che le variabili casuali siano discrete.

Per una variabile casuale X ed un numero reale x, si definisce l'evento $X = x$ come $\{s \in S : X(s) = x\}$; quindi,

$$\Pr\{X = x\} = \sum_{\{s \in S : X(s) = x\}} \Pr\{s\} .$$

La funzione

$$f(x) = \Pr\{X = x\}$$

è la ***funzione di densità*** di una variabile casuale X. Dagli assiomi della probabilità,

$$\Pr\{X = x\} \geq 0 \text{ e } \sum_x \Pr\{X = x\} = 1.$$

Come esempio, si consideri l'esperimento di lanciare un coppia di dadi comuni a sei facce. Vi sono 36 possibili eventi elementari nello spazio campione. Assumendo che la distribuzione di probabilità sia uniforme, cioè ogni evento $s \in S$ sia equamente probabile, si ha che $\Pr\{s\} = 1/36$. Si definisce la variabile casuale X come il *massimo* tra i due valori che i dadi mostrano dopo ogni lancio. Si ha quindi che, per esempio, $\Pr\{X = 3\} = 5/36$, dato che X assegna valore 3 a 5 dei 36 possibili eventi elementari e cioè: (1, 3), (2, 3), (3, 3), (3, 2) e (3, 1).

Spesso si hanno diverse variabili casuali definite sullo stesso spazio campione. Se X e Y sono variabili casuali, allora la funzione

$$f(x, y) = \Pr\{X = x \text{ e } Y = y\}$$

è la ***funzione di densità congiunta*** di X e Y. Per un fissato valore y,

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \quad \text{e} \quad Y = y\} ,$$

ed analogamente, per un fissato valore x,

$$\Pr\{X = x\} = \sum_y \Pr\{X = x \quad \text{e} \quad Y = y\} .$$

Usando la (6.19) della probabilità condizionata, si ha

$$\Pr\{X = x \mid Y = y\} = \frac{\Pr\{X = x \quad \text{e} \quad Y = y\}}{\Pr\{Y = y\}} .$$

Due variabili casuali X e Y si definiscono ***indipendenti*** se per ogni x e y, gli eventi $X = x$ e $Y = y$ sono indipendenti o, equivalentemente, se per ogni x e y, si ha

$$\Pr\{X = x \text{ e } Y = y\} = \Pr\{X = x\}\Pr\{Y = y\}.$$

Dato un insieme di variabili casuali definite sullo stesso spazio campione, si possono definire nuove variabili casuali come somme, prodotti o altre funzioni delle variabili date.

Valore atteso di una variabile casuale

Il più semplice e più utile compendio della distribuzione di una variabile casuale è la "media" dei valori che assume. Il ***valore atteso*** (o ***valore medio***, o ***speranza matematica***) di una variabile casuale discreta X è

$$\mathrm{E}[X] = \sum_x x \Pr\{X = x\} , \tag{6.23}$$

che risulta ben definito se la somma è finita o converge assolutamente. Talvolta la speranza di X è denotata con, μ_X o, quando la variabile è chiara dal contesto, semplicemente con μ.

Si consideri un gioco in cui si lanciano due monete perfette. Si guadagnano 3\$ per ogni testa ma si perdono 2\$ per ogni croce; il valore atteso della variabile casuale X che rappresenta la vincita è

$$\begin{aligned} \mathrm{E}[X] &= 6 \cdot \Pr\{\text{TT}\} + 1 \cdot \Pr\{\text{TC}, \text{CT}\} - 4 \cdot \Pr\{\text{CC}\} \\ &= 6(1/4) + 1(1/2) - 4(1/4) \\ &= 1. \end{aligned}$$

La speranza della somma di due variabili casuali è la somma delle loro speranze

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] , \tag{6.24}$$

ogniqualvolta $\mathrm{E}[X]$ e $\mathrm{E}[Y]$ sono definite. Questa proprietà si estende alle sommatorie finite e assolutamente convergenti di speranze matematiche.

Se X è una generica variabile casuale, qualsiasi funzione $g(x)$ definisce una nuova variabile casuale $g(X)$. Se la speranza di $g(X)$ è definita, allora

$$\mathrm{E}[g(X)] = \sum_x g(x) \Pr\{X = x\} .$$

Ponendo $g(x) = ax$, si ha per ogni costante a.

$$\mathrm{E}[aX] = a\mathrm{E}[X] . \tag{6.25}$$

Di conseguenza, la speranza matematica è lineare: per ogni coppia di variabili casuali X e Y e per ogni costante a

$$\mathrm{E}[aX + Y] = a\mathrm{E}[X] + \mathrm{E}[Y] . \tag{6.26}$$

Quando due variabili casuali X e Y sono indipendenti e ciascuna di esse ha una speranza definita, allora

$$\begin{aligned} \mathrm{E}[XY] &= \sum_x \sum_y xy \Pr\{X = x \quad \text{e} \quad Y = y\} \\ &= \sum_x \sum_y xy \Pr\{X = x\} \Pr\{Y = y\} \\ &= \left(\sum_x x \Pr\{X = x\}\right)\left(\sum_y y \Pr\{Y = y\}\right) \\ &= \mathrm{E}[X]\mathrm{E}[Y] . \end{aligned}$$

In generale, quando n variabili casuali $X_1, X_2, \ldots, X_n$ sono mutuamente indipendenti,

$$\mathrm{E}[X_1 X_2 \cdots X_n] = \mathrm{E}[X_1]\mathrm{E}[X_2] \cdots \mathrm{E}[X_n] . \tag{6.27}$$

Quando una variabile casuale X assume valori sui numeri naturali $\mathrm{N} = \{0, 1, 2, \ldots\}$, vi è una formula semplice per la sua speranza:

$$\begin{aligned} \mathrm{E}[X] &= \sum_{i=0}^{\infty} i \Pr\{X = i\} \\ &= \sum_{i=0}^{\infty} i(\Pr\{X \geq i\} - \Pr\{X \geq i + 1\}) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} . \end{aligned} \tag{6.28}$$

dato che ciascun termine $\Pr\{X \geq i\}$ è addizionato i volte e sottratto $i-1$ volte (eccetto $\Pr\{X \geq 0\}$, che è aggiunto 0 volte e non è mai sottratto).

Varianza e scarto quadratico medio

La *varianza* di una variabile casuale X con speranza $\mathrm{E}[X]$ è

$$\begin{aligned}\mathrm{Var}[X] &= \mathrm{E}\left[(X - \mathrm{E}[X])^2\right] \\ &= \mathrm{E}\left[X^2 - 2X\mathrm{E}[X] + \mathrm{E}^2[X]\right] \\ &= \mathrm{E}\left[X^2\right] - 2\mathrm{E}\left[X\mathrm{E}[X]\right] + \mathrm{E}^2[X] \\ &= \mathrm{E}\left[X^2\right] - 2\mathrm{E}^2[X] + \mathrm{E}^2[X] \\ &= \mathrm{E}\left[X^2\right] - \mathrm{E}^2[X] \ . \end{aligned} \tag{6.29}$$

La giustificazione per le eguaglianze $\mathrm{E}[\mathrm{E}^2[X]] = \mathrm{E}^2[X]$ e $\mathrm{E}[X\mathrm{E}[X]] = \mathrm{E}^2[X]$ è che $\mathrm{E}[X]$ non è una variabile casuale ma semplicemente un numero reale e quindi si può applicare la (6.25) (con $a = \mathrm{E}[X]$). L'equazione (6.29) può essere riscritta per ottenere un'espressione per la speranza di una variabile casuale al quadrato:

$$\mathrm{E}\left[X^2\right] = \mathrm{Var}[X] + \mathrm{E}^2[X] \ . \tag{6.30}$$

La varianza di una variabile casuale X e la varianza di aX sono così correlate:

$$\mathrm{Var}[aX] = a^2\mathrm{Var}[X] \ .$$

Quando X e Y sono variabili casuali indipendenti,

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y] \ .$$

In generale, se n variabili casuali $X_1, X_2, \ldots, X_n$ sono indipendenti a coppie, allora

$$\mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathrm{Var}[X_i] \ . \tag{6.31}$$

Lo ***scarto quadratico medio*** di una variabile casuale X è la radice quadrata positiva della varianza di X. Lo scarto quadratico medio di una variabile casuale X talvolta è denotato con σ_X o semplicemente con σ quando la variabile casuale X è comprensibile dal contesto. Con questa notazione la varianza di X è denotata con σ^2.

Esercizi

6.3-1 Si lancino due comuni dadi a sei facce. Qual è il valore atteso della somma dei due valori che si ottengono con il lancio? Qual è il valore atteso del massimo di tali valori?

6.3-2 Un array $A[1 \ldots n]$ contiene n numeri distinti che sono ordinati casualmente, con ciascuna permutazione degli n elementi equamente probabile. Qual è il valore atteso dell'indice dell'elemento massimo dell'array? Qual è il valore atteso dell'indice dell'elemento minimo dell'array?

6.3-3 Un gioco prevede l'uso di tre dadi in un barattolo. Un giocatore può puntare un dollaro su un qualsiasi numero da 1 a 6. Il barattolo è agitato e la conclusione è la seguente: se il numero scelto dal giocatore non appare su nessun dado, allora perde il dollaro; altrimenti, se il suo numero compare esattamente su k dei tre dadi, per $k = 1, 2, 3$, egli tiene il suo dollaro e vince altri k dollari. Qual è la speranza matematica del guadagno del giocatore, provando una sola volta?

* *6.3-4* Siano X e Y due variabili casuali indipendenti. Provare che $f(X)$ e $g(Y)$ sono indipendenti per ogni scelta delle funzioni f e g.

* *6.3-5* Sia X una variabile casuale non negativa, e si supponga che $\mathrm{E}[X]$ sia ben definita. Provare la ***disuguaglianza di Markov***

$$\Pr\{X \geq t\} \leq \mathrm{E}[X]/t \tag{6.32}$$

per ogni $t > 0$.

* *6.3-6* Sia S uno spazio campione e siano X e X' variabili casuali tali che $X(s) \geq X'(s)$ per ogni $s \in S$. Provare che per ogni costante reale t,

$$\Pr\{X \geq t\} \geq \Pr\{X' \geq t\} \ .$$

6.3-7 Qual è più grande: la speranza matematica del quadrato di una variabile casuale o il quadrato della sua speranza?

6.3-8 Mostrare che, per qualsiasi variabile casuale X che assume solo i valori 0 e 1, $\mathrm{Var}[X] = \mathrm{E}[X]\,\mathrm{E}[1 - X]$

6.3-9 Provare che $\mathrm{Var}[aX] = a^2\mathrm{Var}[X]$ dalla definizione (6.29) della varianza.

6.4 Distribuzione geometrica e distribuzione binomiale

Il lancio di una moneta è un esempio di ***prova di Bernoulli***, che è definita come un esperimento con due soli possibili risultati: ***successo***, che si verifica con probabilità p e ***fallimento***, che si verifica con probabilità $q = 1 - p$. Quando si parla di ***prove di Bernoulli*** nel loro complesso, si intende che gli esperimenti sono mutuamente indipendenti e che, a meno che non sia diversamente specificato, ognuno ha la stessa probabilità p di successo. Due distribuzioni importanti derivano dalle prove di Bernoulli: la distribuzione geometrica e la distribuzione binomiale.

La distribuzione geometrica

Si supponga di avere una sequenza di prove di Bernoulli, ognuna con probabilità p di successo e probabilità $q = 1 - p$ di fallimento. Quante prove occorre fare prima d'ottenere un successo? Sia la variabile casuale X il numero di prove necessarie per ottenere un successo.

Figura 6.1 *Una distribuzione geometrica con probabilità di successo $p = 1/3$ e probabilità di fallimento $q = 1 - p$. La speranza della distribuzione è $1/p = 3$.*

Allora X assume valori nell'intervallo $\{1, 2, \ldots\}$, e per $k > 1$ vale che

$$\Pr\{X = k\} = q^{k-1}p \ , \tag{6.33}$$

perché prima di un successo si hanno $k - 1$ fallimenti. Una distribuzione di probabilità che soddisfi l'equazione (6.33) è detta ***distribuzione geometrica***. La figura 6.1 illustra tale distribuzione.

Assumendo $p < 1$, la speranza di una distribuzione geometrica può essere calcolata usando l'identità (3.6):

$$\begin{aligned} \mathrm{E}[X] &= \sum_{k=1}^{\infty} kq^{k-1}p \\ &= \frac{p}{q}\sum_{k=0}^{\infty} kq^k \\ &= \frac{p}{q}\cdot\frac{q}{(1-q)^2} \\ &= 1/p \ . \end{aligned} \tag{6.34}$$

Di conseguenza, in media, occorrono $1/p$ prove prima di ottenere un successo, che è anche la risposta intuitiva. La varianza, che può essere calcolata analogamente, è

$$\mathrm{Var}[X] = q/p^2 \ . \tag{6.35}$$

Figura 6.2 *La distribuzione binomiale $b(k; 15, 1/3)$ che si ottiene con $n = 15$ prove di Bernoulli, ciascuna con probabilità di successo $p = 1/3$. La speranza della distribuzione è $np = 5$.*

Per esempio si supponga di lanciare ripetutamente due dadi finché si ottenga un sette o un undici. Dei 36 possibili risultati, 6 valgono sette e 2 valgono undici. Di conseguenza la probabilità di successo è $p = 8/36 = 2/9$ e si devono lanciare i dadi $1/p = 9/2 = 4.5$ volte, in media.

La distribuzione binomiale

Quanti successi si verificano durante n prove di Bernoulli, dove un successo si verifica con probabilità p e un fallimento con probabilità $q = 1 - p$? Si definisca la variabile casuale X come il numero di successi su n prove: quindi X assume valori nell'intervallo $\{0, 1, \ldots, n\}$ e per $k = 0, \ldots, n$,

$$\Pr\{X = k\} = \binom{n}{k}p^k q^{n-k} \ , \tag{6.36}$$

perché vi sono $\binom{n}{k}$ modi di scegliere quali k delle n prove sono successi, e la probabilità che ognuna di esse si verifichi è $p^k q^{n-k}$. Una distribuzione di probabilità che soddisfi l'equazione (6.36) è detta ***distribuzione binomiale***. Per comodità, si definisce la famiglia di distribuzioni binomiali usando la notazione

$$b(k; n, p) = \binom{n}{k}p^k(1-p)^{n-k} \ . \tag{6.37}$$

La figura 6.2 illustra una distribuzione binomiale. Il nome "binomiale" ha origine dal fatto che (6.37) è il k-esimo termine dello sviluppo di $(p + q)^n$. Di conseguenza, dato che $p + q = 1$, si ha

$$\sum_{k=0}^{n} b(k; n, p) = 1 \ , \tag{6.38}$$

come richiesto dall'assioma 2 degli assiomi della probabilità.

Si può calcolare la speranza di una variabile con distribuzione binomiale dalle equazioni (6.14) e (6.38). Sia X una variabile casuale che segue la distribuzione binomiale $b(k; n, p)$, e sia $q = 1 - p$. Dalla definizione di speranza, si ha

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{k=0}^{n} k\, b(k; n, p) \\
&= \sum_{k=1}^{n} k \binom{n}{k} p^k q^{n-k} \\
&= np \sum_{k=1}^{n} \binom{n-1}{k-1} p^{k-1} q^{n-k} \\
&= np \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{(n-1)-k} \\
&= np \sum_{k=0}^{n-1} b(k; n-1, p) \\
&= np\,. \qquad (6.39)
\end{aligned}$$

Usando la linearità della speranza, si può ottenere lo stesso risultato con una quantità sostanzialmente inferiore di calcoli algebrici. Sia X_i la variabile casuale che descrive il numero di successi dell'i-esima prova. Allora $\mathrm{E}[X_i] = p \cdot 1 + q \cdot 0 = p$ e, per la linearità della speranza (6.26), il numero atteso di successi su n prove è

$$\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] \\
&= \sum_{i=1}^{n} p \\
&= np\,.
\end{aligned}$$

Lo stesso approccio può essere usato per calcolare la varianza della distribuzione. Usando l'equazione (6.29), si ha $\mathrm{Var}[X_i] = \mathrm{E}[X_i^2] - \mathrm{E}^2[X_i]$. Poiché X_i può assumere solo valori 0 e 1, si ha $\mathrm{E}[X_i^2] = \mathrm{E}[X_i] = p$, e quindi

$$\mathrm{Var}[X_i] = p - p^2 = pq\,. \qquad (6.40)$$

Per calcolare la varianza di X, si sfrutta l'indipendenza delle n prove; quindi, dall'equazione (6.31),

$$\begin{aligned}
\mathrm{Var}[X] &= \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] \\
&= \sum_{i=1}^{n} \mathrm{Var}[X_i] \\
&= \sum_{i=1}^{n} pq \\
&= npq\,. \qquad (6.41)
\end{aligned}$$

Come si può vedere dalla figura 6.2, la distribuzione binomiale $b(k; n, p)$, per k che va da 0 a n, cresce finché k raggiunge il valore medio np e poi decresce. Si può provare che la distribuzione si comporta sempre in questo modo osservando il rapporto di termini successivi:

$$\begin{aligned}
\frac{b(k; n, p)}{b(k-1; n, p)} &= \frac{\binom{n}{k} p^k q^{n-k}}{\binom{n}{k-1} p^{k-1} q^{n-k+1}} \\
&= \frac{n!(k-1)!(n-k+1)!p}{k!(n-k)!n!q} \\
&= \frac{(n-k+1)p}{kq} \\
&= 1 + \frac{(n+1)p - k}{kq}\,. \qquad (6.42)
\end{aligned}$$

Questo rapporto è più grande di 1 esattamente quando $(n+1)p - k$ è positivo. Di conseguenza, $b(k; n, p) > b(k-1; n, p)$ per $k < (n+1)p$ (la distribuzione cresce), mentre $b(k; n, p) < b(k-1; n, p)$ per $k > (n+1)p$ (la distribuzione decresce). Se $k = (n+1)p$ è un intero, allora $b(k; n, p) = b(k-1; n, p)$, e così la distribuzione ha due massimi: per $k = (n+1)p$ e per $k - 1 = (n+1)p - 1 = np - q$; altrimenti, raggiunge un massimo per l'unico intero k che sta nell'intervallo $np - q < k < (n+1)p$.

Il seguente lemma fornisce un limite superiore per la distribuzione binomiale.

Lemma 6.1

Siano $n \geq 0$, $0 < p < 1$, $q = 1 - p$ e $0 \leq k \leq n$. Allora

$$b(k; n, p) \leq \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}\,.$$

Dimostrazione. Usando l'equazione (6.10), si ha

$$\begin{aligned}
b(k; n, p) &= \binom{n}{k} p^k q^{n-k} \\
&\leq \left(\frac{n}{k}\right)^k \left(\frac{n}{n-k}\right)^{n-k} p^k q^{n-k} \\
&= \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k}\,.
\end{aligned}$$

■

Esercizi

6.4-1 Verificare l'assioma 2 della probabilità per la distribuzione geometrica.

6.4-2 Quante volte in media si devono lanciare 6 monete perfette prima di ottenere 3 teste e 3 croci?

6.4-3 Dimostrare che $b(k; n, p) = b(n-k; n, q)$, dove $q = 1 - p$.

6.4-4 Dimostrare che il valore massimo della distribuzione binomiale $b(k; n, p)$ è approssimativamente $1/\sqrt{2\pi npq}$, dove $q = 1 - p$.

* *6.4-5* Dimostrare che la probabilità di non avere successo in n prove di Bernoulli, ognuna con probabilità $p = 1/n$, è approssimativamente $1/e$. Dimostrare che anche la probabilità di ottenere esattamente un successo è $1/e$.

* *6.4-6* Il professor Rosencrantz lancia una moneta perfetta n volte, e così fa anche il professor Guildenstern. Dimostrare che la probabilità che essi ottengano lo stesso numero di teste è $\binom{2n}{n}/4^n$. (*Suggerimento*: per il professor Rosencrantz è un successo ottenere una testa; per il professor Guildenstern è un successo ottenere una croce). Utilizzando lo stesso schema di dimostrazione verificare l'identità:

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n} .$$

* *6.4-7* Mostrare che per $0 \le k \le n$,

$$b(k; n, 1/2) \le 2^{n\,H(k/n)-n} ,$$

dove $H(x)$ è la funzione di entropia (6.13).

* *6.4-8* Considerare n prove di Bernoulli, dove per $i = 1, 2, \ldots, n$, l'i-esima prova ha probabilità p_i di successo, e sia X la variabile casuale che denota il numero totale di successi. Sia $p \ge p_i$ per ciascun $i = 1, 2, \ldots, n$. Provare che per $1 \le k \le n$,

$$\Pr\{X < k\} \le \sum_{i=0}^{k-1} b(i; n, p) .$$

* *6.4-9* Sia X la variabile casuale che denota il numero totale di successi in un insieme A di n prove di Bernoulli, dove l'i-esima prova ha probabilità di successo p_i, e sia X' la variabile casuale che denota il numero totale di successi in un secondo insieme A' di n prove di Bernoulli, dove l'i-esima prova ha probabilità di successo $p'_i \ge p_i$. Provare che per $0 \le k \le n$

$$\Pr\{X' \ge k\} \ge \Pr\{X \ge k\} .$$

(*Suggerimento*: mostrare come ottenere le prove di Bernoulli di A' da un esperimento che coinvolga le prove di A, e usare il risultato dell'Esercizio 6.3-6).

* 6.5 Code della distribuzione binomiale

La probabilità di avere almeno, o al più, k successi in n prove di Bernoulli, ognuna con probabilità p di successo, è spesso di maggior interesse della probabilità di avere esattamente k successi. In questo paragrafo, si analizzeranno le ***code*** della distribuzione binomiale: le due regioni della distribuzione $b(k; n, p)$ lontane dal valore medio np. Si proveranno alcuni importanti limiti su (la somma di tutti i termini di) una coda.

Innanzitutto si fornirà un limite sulla coda destra della distribuzione $b(k; n, p)$. I limiti sulla coda sinistra possono essere determinati invertendo i ruoli di *successo* e *fallimento*.

Teorema 6.2

Si consideri una sequenza di n prove di Bernoulli, dove un successo si verifica con probabilità p. Sia X la variabile casuale che denota il numero totale di successi. Allora per $0 \le k \le n$, la probabilità di almeno k successi è:

$$\begin{aligned} \Pr\{X \ge k\} &= \sum_{i=k}^{n} b(i; n, p) \\ &\le \binom{n}{k} p^k . \end{aligned}$$

Dimostrazione. Facendo uso della disuguaglianza (6.15)

$$\binom{n}{k+i} \le \binom{n}{k}\binom{n-k}{i}$$

si ha

$$\begin{aligned} \Pr\{X \ge k\} &= \sum_{i=k}^{n} b(i; n, p) \\ &= \sum_{i=0}^{n-k} b(k+i; n, p) \\ &= \sum_{i=0}^{n-k} \binom{n}{k+i} p^{k+i}(1-p)^{n-(k+i)} \\ &\le \sum_{i=0}^{n-k} \binom{n}{k}\binom{n-k}{i} p^{k+i}(1-p)^{n-(k+i)} \\ &= \binom{n}{k} p^k \sum_{i=0}^{n-k} \binom{n-k}{i} p^i (1-p)^{(n-k)-i} \\ &= \binom{n}{k} p^k \sum_{i=0}^{n-k} b(i; n-k, p) \\ &= \binom{n}{k} p^k , \end{aligned}$$

dato che $\sum_{i=0}^{n-k} b(i; n-k, p) = 1$ dall'equazione (6.38). ■

Il seguente corollario riformula il teorema per la coda sinistra della distribuzione binomiale. In generale, si lascia al lettore l'adattamento dei limiti da una coda all'altra.

Corollario 6.3

Si consideri una sequenza di n prove di Bernoulli, dove un successo si verifica con probabilità p. Sia X la variabile casuale che denota il numero totale di successi, allora per $0 \le k \le n$, la probabilità di avere al più k successi è

$$\begin{aligned}\Pr\{X \le k\} &= \sum_{i=0}^{k} b(i;n,p) \\ &\le \binom{n}{n-k}(1-p)^{n-k} \\ &= \binom{n}{k}(1-p)^{n-k} .\end{aligned}$$

■

Il nostro prossimo limite si concentra sulla coda sinistra della distribuzione binomiale. Lontano dal valore medio, il numero di successi nella coda sinistra diminuisce esponenzialmente, come dimostra il seguente teorema.

Teorema 6.4

Si consideri una sequenza di n prove di Bernoulli, dove un successo si verifica con probabilità p e un fallimento con probabilità $q = 1 - p$. Sia X la variabile casuale che denota il numero totale di successi. Allora, per $0 < k < np$, la probabilità di avere meno di k successi è:

$$\begin{aligned}\Pr\{X < k\} &= \sum_{i=0}^{k-1} b(i;n,p) \\ &< \frac{kq}{np-k}\, b(k;n,p) .\end{aligned}$$

Dimostrazione. Si limita la serie $\sum_{i=0}^{k-1} b(i;n,p)$ con una serie geometrica usando la tecnica presentata nel paragrafo 3.2. Per $i = 1, 2, \ldots, k$, si ha dall'equazione (6.42)

$$\begin{aligned}\frac{b(i-1;n,p)}{b(i;n,p)} &= \frac{iq}{(n-i+1)p} \\ &< \left(\frac{i}{n-i}\right)\left(\frac{q}{p}\right) \\ &\le \left(\frac{k}{n-k}\right)\left(\frac{q}{p}\right) .\end{aligned}$$

Ponendo

$$x = \left(\frac{k}{n-k}\right)\left(\frac{q}{p}\right) < 1 ,$$

segue che

$$b(i-1;n,p) < x\, b(i;n,p)$$

per $0 < i \le k$. Iterando si ottiene

$$b(i;n,p) < x^{k-i}\, b(k;n,p)$$

per $0 \le i < k$, e quindi

$$\begin{aligned}\sum_{i=0}^{k-1} b(i;n,p) &< \sum_{i=0}^{k-1} x^{k-i} b(k;n,p) \\ &< b(k;n,p)\sum_{i=1}^{\infty} x^i \\ &= \frac{x}{1-x}\, b(k;n,p) \\ &= \frac{kq}{np-k}\, b(k;n,p) .\end{aligned}$$

■

Quando $k \le np/2$, si ha $kq/(np-k) \le 1$, il cui significato è che $b(k;n,p)$ limita la somma di tutti i termini più piccoli di k. Per esempio, si supponga di lanciare n monete; usando $p = 1/2$ e $k = n/4$, il Teorema 6.4 dice che la probabilità di ottenere meno di $n/4$ teste è minore della probabilità di ottenere esattamente $n/4$ teste. Inoltre, per ogni $r \ge 4$, la probabilità di ottenere meno di n/r teste è minore della probabilità di ottenere esattamente n/r teste. Il Teorema 6.4 può anche essere abbastanza utile insieme con i limiti superiori della distribuzione binomiale, come il Lemma 6.1.

Un limite per la coda di destra può essere determinato in modo simile.

Corollario 6.5

Si consideri una sequenza di n prove di Bernoulli, dove un successo si verifica con probabilità p. Sia X la variabile casuale che denota il numero totale di successi, allora per $np < k < n$, la probabilità di ottenere più di k successi è

$$\begin{aligned}\Pr\{X > k\} &= \sum_{i=k+1}^{n} b(i;n,p) \\ &< \frac{(n-k)p}{k-np}\, b(k;n,p) .\end{aligned}$$

■

Il prossimo teorema considera n prove di Bernoulli, ciascuna con probabilità p_i di successo, per $i = 1, 2, \ldots, n$. Il corollario che lo segue mostra come si possa usare tale teorema per fornire un limite alla coda destra di una distribuzione binomiale ponendo $p_i = p$ per ciascuna prova.

Teorema 6.6

Si consideri una sequenza di n prove di Bernoulli, dove nella i-esima prova, per $i = 1, 2, \ldots, n$, il successo si verifica con probabilità p_i ed il fallimento con probabilità $q_i = 1 - p_i$. Sia X la variabile casuale che descrive il numero totale di successi e sia $\mu = \mathrm{E}[X]$.

Allora per $r > \mu$,

$$\Pr\{X - \mu \ge r\} \le \left(\frac{\mu e}{r}\right)^r .$$

Dimostrazione. Dato che per ogni $\alpha > 0$ la funzione $e^{\alpha x}$ è strettamente crescente in x,

$$\Pr\{X - \mu \geq r\} = \Pr\{e^{\alpha(X-\mu)} \geq e^{\alpha r}\} ,$$

dove α sarà determinato in seguito. Usando la disuguaglianza di Markov (6.32), si ottiene

$$\Pr\{X - \mu \geq r\} \leq \mathrm{E}\left[e^{\alpha(X-\mu)}\right] e^{-\alpha r} . \tag{6.43}$$

La parte più importante della dimostrazione consiste nel limitare $\mathrm{E}[e^{\alpha(X-\mu)}]$ e nel sostituire ad α un valore opportuno nella disuguaglianza (6.43). Prima si valuterà $\mathrm{E}[e^{\alpha(X-\mu)}]$. Per $i = 1, 2, \ldots, n$, sia X_i la variabile casuale che è 1 se l'i-esima prova di Bernoulli è un successo e 0 se è un fallimento. Perciò,

$$X = \sum_{i=1}^{n} X_i$$

e

$$X - \mu = \sum_{i=1}^{n} (X_i - p_i) .$$

Sostituendo $X - \mu$, si ottiene

$$\begin{aligned} \mathrm{E}\left[e^{\alpha(X-\mu)}\right] &= \mathrm{E}\left[\prod_{i=1}^{n} e^{\alpha(X_i - p_i)}\right] \\ &= \prod_{i=1}^{n} \mathrm{E}\left[e^{\alpha(X_i - p_i)}\right] , \end{aligned}$$

che segue dalla (6.27), perché la mutua indipendenza delle variabili casuali X_i implica la mutua indipendenza delle variabili casuali $e^{\alpha(X_i - p_i)}$ (si veda l'Esercizio 6.3-4). Per definizione di speranza,

$$\begin{aligned} \mathrm{E}\left[e^{\alpha(X_i - p_i)}\right] &= e^{\alpha(1-p_i)} p_i + e^{\alpha(0-p_i)} q_i \\ &= p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \\ &\leq p_i e^{\alpha} + 1 \\ &\leq \exp(p_i e^{\alpha}) , \end{aligned} \tag{6.44}$$

dove $\exp(x)$ denota la funzione esponenziale: $\exp(x) = e^x$. (La disuguaglianza (6.44) segue dalle disuguaglianze $\alpha > 0$, $q_i \leq 1$, $e^{\alpha q_i} \leq e^{\alpha}$, e $e^{-\alpha p_i} \leq 1$ e l'ultima linea segue dalla disuguaglianza (2.7)). Di conseguenza,

$$\begin{aligned} \mathrm{E}\left[e^{\alpha(X-\mu)}\right] &\leq \prod_{i=1}^{n} \exp(p_i e^{\alpha}) \\ &= \exp(\mu e^{\alpha}) , \end{aligned}$$

dato che $\mu = \sum_{i=1}^{n} p_i$. Quindi, dalla disuguaglianza (6.43), segue che

$$\Pr\{X - \mu \geq r\} \leq \exp(\mu e^{\alpha} - \alpha r) . \tag{6.45}$$

Scegliendo $\alpha = \ln(r/\mu)$ (si veda l'Esercizio 6.5-6), si ottiene

$$\begin{aligned} \Pr\{X - \mu \geq r\} &\leq \exp(\mu e^{\ln(r/\mu)} - r\ln(r/\mu)) \\ &= \exp(r - r\ln(r/\mu)) \\ &= \frac{e^r}{(r/\mu)^r} \\ &= \left(\frac{\mu e}{r}\right)^r . \end{aligned}$$

■

Quando si considerano prove di Bernoulli in cui ciascuna prova ha la stessa probabilità di successo, allora dal Teorema 6.6 deriva il seguente corollario che definisce un limite per la coda destra di una distribuzione binomiale.

Corollario 6.7

Si consideri una sequenza di n prove di Bernoulli, dove un successo si verifica con probabilità p ed un fallimento con probabilità $q = 1 - p$. Allora, per $r > np$,

$$\begin{aligned} \Pr\{X - np \geq r\} &= \sum_{k=\lceil np+r \rceil}^{n} b(k; n, p) \\ &\leq \left(\frac{npe}{r}\right)^r . \end{aligned}$$

Dimostrazione. Per una distribuzione binomiale l'equazione (6.39) implica che $\mu = \mathrm{E}[X] = np$.

■

Esercizi

* *6.5-1* Che cosa è meno probabile: non ottenere alcuna testa quando si lancia una moneta perfetta n volte o ottenere meno di n teste quando si lancia la moneta $4n$ volte?

* *6.5-2* Dimostrare che

$$\sum_{i=0}^{k-1} \binom{n}{i} a^i < (a+1)^n \frac{k}{na - k(a+1)} b(k; n, a/(a+1))$$

per ogni $a > 0$ e per ogni k tale che $0 < k < n$.

* *6.5-3* Provare che se $0 < k < np$, $0 < p < 1$ e $q = 1 - p$, allora

$$\sum_{i=0}^{k-1} p^i q^{n-i} < \frac{kq}{np - k} \left(\frac{np}{k}\right)^k \left(\frac{nq}{n-k}\right)^{n-k} .$$

* *6.5-4* Dimostrare che le condizioni del Teorema 6.6 implicano che

$$\Pr\{\mu - X \geq r\} \leq \left(\frac{(n-\mu)e}{r}\right)^r .$$

Analogamente, mostrare che le condizioni del Corollario 6.7 implicano che

$$\Pr\{np - X \geq r\} \leq \left(\frac{nqe}{r}\right)^r .$$

* *6.5-5* Si consideri una sequenza di n prove di Bernoulli, dove nella i-esima prova, per $i = 1, 2, \ldots, n$, il successo si verifica con probabilità p_i ed il fallimento con probabilità $q_i = 1 - p_i$. Sia X la variabile casuale che descrive il numero totale dei successi e sia $\mu = \mathrm{E}[X]$. Allora per $r \geq 0$,

$$\Pr\{X - \mu \geq r\} \leq e^{-r^2/2n} .$$

(*Suggerimento*: provare che $p_i e^{\alpha q_i} + q_i e^{-\alpha p_i} \leq e^{-\alpha^2/2}$. Quindi applicare la tecnica seguita per dimostrare il Teorema 6.6, usando questa disuguaglianza al posto della disuguaglianza (6.44)).

* *6.5-6* Mostrare che scegliendo $\alpha = \ln(r/\mu)$ si minimizza la parte destra della disuguaglianza (6.45).

6.6 Analisi probabilistica

Questo paragrafo usa tre esempi per illustrare l'analisi probabilistica. Il primo determina la probabilità che, in una stanza con k persone, alcune coppie di persone abbiano lo stesso compleanno. Il secondo esempio considera il lancio casuale di palline dentro dei contenitori. Il terzo esamina sequenze di teste consecutive nel lancio di una moneta.

6.6.1 Il paradosso del compleanno

Un buon esempio per illustrare il ragionamento probabilistico è il classico ***paradosso del compleanno***. Quante persone devono esserci in una stanza perché ci sia una buona probabilità che due di loro siano nate nello stesso giorno dell'anno? Poche, è la sorprendente risposta. Il paradosso è che in realtà ne bastano molte di meno del numero dei giorni di un anno, come si vedrà di seguito.

Per rispondere alla domanda, si indichino le persone nella stanza con gli interi $1, 2, \ldots, k$ dove k è il numero delle persone nella stanza. Si ignori il problema degli anni bisestili e si assuma che ogni anno abbia $n = 365$ giorni. Per $i = 1, 2, \ldots, k$, sia b_i il giorno dell'anno in cui cade l'i-esimo compleanno, con $1 \leq b_i \leq n$. Si assuma anche che i compleanni siano uniformemente distribuiti sugli n giorni dell'anno, così che $\Pr\{b_i = r\} = 1/n$ per $i = 1, 2, \ldots, k$ ed $r = 1, 2, \ldots, n$.

La probabilità che due persone i e j facciano il compleanno lo stesso giorno, dipende dal fatto che la selezione casuale dei compleanni sia indipendente. Se i compleanni sono indipendenti, allora la probabilità che il compleanno di i e quello di j cadano entrambi il giorno r è

$$\begin{aligned}\Pr\{b_i = r \text{ e } b_j = r\} &= \Pr\{b_i = r\}\Pr\{b_j = r\} \\ &= 1/n^2 .\end{aligned}$$

Quindi, la probabilità che entrambi i compleanni cadano lo stesso giorno è

$$\begin{aligned}\Pr\{b_i = b_j\} &= \sum_{r=1}^{n} \Pr\{b_i = r \text{ e } b_j = r\} \\ &= \sum_{r=1}^{n} (1/n^2) \\ &= 1/n .\end{aligned}$$

Più intuitivamente, una volta che b_i sia stato scelto, la probabilità che b_j venga scelto uguale è $1/n$. Perciò, la probabilità che i e j abbiano lo stesso compleanno è la stessa probabilità che il compleanno di uno di essi cada in un dato giorno. Attenzione però perché questa coincidenza dipende dall'assunzione che i compleanni siano indipendenti.

Si può analizzare la probabilità che almeno 2 persone su k abbiano compleanni coincidenti osservando l'evento complementare. La probabilità che almeno due compleanni siano uguali è 1 diminuito della probabilità che tutti i compleanni siano differenti. L'evento che k persone abbiano compleanni distinti è

$$B_k = \bigcap_{i=1}^{k-1} A_i ,$$

dove A_i è l'evento che il compleanno della persona $(i + 1)$ è diverso da quello della persona j per ogni $j \leq i$, cioè,

$$A_i = \{b_{i+1} \neq b_j : j = 1, 2, \ldots, i\} .$$

Poiché si può scrivere $B_k = A_{k-1} \cap B_{k-1}$, si ottiene dall'equazione (6.20) la ricorrenza:

$$\Pr\{B_k\} = \Pr\{B_{k-1}\}\Pr\{A_{k-1} \mid B_{k-1}\} , \tag{6.46}$$

dove si assume come condizione iniziale $\Pr\{B_1\} = 1$. In altre parole, la probabilità che $b_1, b_2, \ldots, b_k$ siano compleanni distinti è la probabilità che $b_1, b_2, \ldots, b_{k-1}$ siano compleanni distinti moltiplicata per la probabilità che $b_k \neq b_i$ per $i = 1, 2, \ldots, k-1$, dato che $b_1, b_2, \ldots, b_{k-1}$ sono distinti.

Se $b_1, b_2, \ldots, b_{k-1}$ sono distinti, la probabilità condizionata che $b_k \neq b_i$ per $i = 1, 2, \ldots, k-1$ è $(n - k + 1)/n$, dato che su n giorni, vi sono $n - (k-1)$ giorni che non vengono presi. Iterando la ricorrenza (6.46) si ottiene

$$\begin{aligned}\Pr\{B_k\} &= \Pr\{B_1\}\Pr\{A_1 \mid B_1\}\Pr\{A_2 \mid B_2\}\cdots\Pr\{A_{k-1} \mid B_{k-1}\} \\ &= 1 \cdot \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)\cdots\left(\frac{n-k+1}{n}\right) \\ &= 1 \cdot \left(1 - \frac{1}{n}\right)\left(1 - \frac{2}{n}\right)\cdots\left(1 - \frac{k-1}{n}\right) .\end{aligned}$$

La disuguaglianza (2.7), $1 + x \leq e^x$, fornisce

$$\begin{aligned}\Pr\{B_k\} &\leq e^{-1/n} e^{-2/n} \cdots e^{-(k-1)/n} \\ &= e^{-\sum_{i=1}^{k-1} i/n} \\ &= e^{-k(k-1)/2n} \\ &\leq 1/2\end{aligned}$$

quando $-k(k-1)/2n \le \ln(1/2)$. La probabilità che tutti i k compleanni siano distinti è al più 1/2 quando $k(k-1) \ge 2n\ln 2$ oppure, risolvendo l'equazione quadratica, quando $k \ge (1+\sqrt{1+(8\ln 2)n})/2$. Per $n = 365$, si deve avere $k \ge 23$. Perciò, se almeno 23 persone sono in una stanza, la probabilità che almeno due di esse abbiano lo stesso compleanno è almeno 1/2. Su Marte, un anno è lungo 669 giorni marziani; quindi occorrono 31 marziani per ottenere lo stesso risultato.

Un altro metodo di analisi

Si può usare la linearità della speranza (equazione (6.26)) per fornire un'analisi più semplice anche se approssimativa del paradosso del compleanno. Per ogni coppia (i, j) di k persone nella stanza, si definisca la variabile casuale X_{ij} per $1 \le i < j \le k$, come

$$X_{ij} = \begin{cases} 1 & \text{se la persona } i \text{ e la persona } j \text{ hanno lo stesso compleanno} \\ 0 & \text{altrimenti.} \end{cases}$$

La probabilità che due persone abbiano lo stesso compleanno è $1/n$, quindi per la definizione di speranza (6.23),

$$\begin{aligned} \mathrm{E}[X_{ij}] &= 1 \cdot (1/n) + 0 \cdot (1 - 1/n) \\ &= 1/n . \end{aligned}$$

Il numero atteso di coppie di individui con lo stesso compleanno è, dall'equazione (6.24), esattamente la somma delle speranze matematiche individuali delle coppie, che è

$$\begin{aligned} \sum_{i=2}^{k} \sum_{j=1}^{i-1} \mathrm{E}[X_{ij}] &= \binom{k}{2} \frac{1}{n} \\ &= \frac{k(k-1)}{2n} . \end{aligned}$$

Perciò, quando $k(k-1) \ge 2n$, il numero atteso di coppie di compleanni è almeno 1. Quindi, se vi sono almeno $\sqrt{2n}$ individui in una stanza ci si può aspettare che almeno due abbiano lo stesso compleanno. Per $n = 365$, se $k = 28$ il numero atteso di coppie con lo stesso compleanno è $(28 \cdot 27)/(2 \cdot 365) \approx 1{,}0356$. Quindi, con almeno 28 persone, ci si aspetta di trovare almeno una coppia di compleanni uguali. Su Marte, dove un anno è lungo 669 giorni marziani, si ha bisogno di almeno 38 marziani.

La prima analisi ha determinato il numero di persone necessarie affinché la probabilità che esista una coppia di compleanni uguali ecceda 1/2, mentre la seconda analisi ha determinato il numero di persone per cui il numero atteso di compleanni uguali sia 1. Sebbene il numero di persone differisca nelle due situazioni, asintoticamente è lo stesso: $\Theta(\sqrt{n})$.

6.6.2 Palline e contenitori

Si consideri il processo del lancio casuale di palline identiche, numerate da 1 a b, in b contenitori. I lanci sono indipendenti, e ogni pallina può finire in un qualunque contenitore con la stessa probabilità. La probabilità che una pallina finisca in un dato contenitore è $1/b$. Quindi, il processo del lancio della pallina è una sequenza di prove di Bernoulli con una probabilità $1/b$ di successo, dove il successo significa che la pallina cade in un dato contenitore. Si possono porre parecchie domande interessanti sul processo del lancio.

Quante palline finiscono in un dato contenitore? Il numero di palline che cadono in un dato contenitore segue la distribuzione binomiale $b(k; n, 1/b)$. Se vengono lanciate n palline, il numero atteso di palline che cadono in un dato contenitore è n/b.

Quante palline si devono lanciare, in media, perché un dato contenitore contenga una pallina? Il numero di lanci perché un dato contenitore riceva una pallina segue la distribuzione geometrica con probabilità $1/b$ e quindi il numero atteso di lanci per ottenere un successo è $1/(1/b) = b$.

Quante palline si devono lanciare perché ogni contenitore contenga almeno una pallina? Sia "centro" il nome dato ad un lancio per cui la pallina finisce in un contenitore vuoto. Si vuole conoscere la media del numero n di lanci richiesti per ottenere b centri.

I centri possono essere usati per partizionare gli n lanci in tappe. L'i-esima tappa consiste dei lanci tra l'$(i-1)$-esimo e l'i-esimo centro. La prima tappa è costituita dal primo lancio, dato che si è sicuri di avere un centro quando tutti i contenitori sono vuoti. Per ogni lancio durante l'i-esima tappa ci sono $i-1$ contenitori che contengono palline e $b-i+1$ contenitori vuoti. Quindi per ogni lancio dell'i-esima tappa, la probabilità di ottenere un centro è $(b-i+1)/b$.

Sia n_i il numero dei lanci dell'i-esima tappa. Quindi, il numero di lanci richiesti per avere b centri è: $n = \sum_{i=1}^{b} n_i$. Ogni variabile casuale n_i ha una distribuzione geometrica con probabilità di successo $(b-i+1)/b$, e perciò

$$\mathrm{E}[n_i] = \frac{b}{b-i+1} .$$

Per la linearità della speranza,

$$\begin{aligned} \mathrm{E}[n] &= \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] \\ &= \sum_{i=1}^{b} \mathrm{E}[n_i] \\ &= \sum_{i=1}^{b} \frac{b}{b-i+1} \\ &= b \sum_{i=1}^{b} \frac{1}{i} \\ &= b(\ln b + O(1)) . \end{aligned}$$

L'ultima riga segue dal limite (3.5) sulle serie armoniche. In conclusione occorrono approssimativamente $b \ln b$ lanci prima che ci si possa aspettare che ogni contenitore abbia una pallina.

6.6.3 Sequenze di successi

Si supponga di lanciare n volte una moneta perfetta. Qual è la sequenza di *teste* più lunga che ci si può aspettare? La risposta è $\Theta(\lg n)$, come mostra l'analisi seguente.

Si comincia provando che la lunghezza attesa della sequenza di teste più lunga è $O(\lg n)$. Sia A_{ik} l'evento per cui una sequenza di teste lunga almeno k cominci con l'i-esimo lancio di moneta o, più precisamente, l'evento che i k lanci consecutivi $i, i+1, \ldots, i+k-1$ producano solo teste, dove $1 \le k \le n$ e $1 \le i \le n-k+1$. Per ogni dato evento A_{ik}, la probabilità che tutti i lanci siano teste ha una distribuzione geometrica con $p = q = 1/2$:

$$\Pr\{A_{ik}\} = 1/2^k . \tag{6.47}$$

Per $k = 2\lceil \lg n \rceil$,

$$\begin{aligned} \Pr\{A_{i,2\lceil \lg n\rceil}\} &= 1/2^{2\lceil \lg n\rceil} \\ &\le 1/2^{2\lg n} \\ &= 1/n^2 , \end{aligned}$$

e così la probabilità che una sequenza di teste lunga almeno $\lceil 2\lg n \rceil$ cominci in posizione i è sufficientemente piccola, specialmente considerando che ci sono al più n posizioni (in realtà $n - 2\lceil \lg n \rceil + 1$) da cui la sequenza può cominciare. La probabilità che una sequenza di teste di lunghezza almeno $\lceil 2\lg n \rceil$ cominci da una qualsiasi posizione, è perciò

$$\begin{aligned} \Pr\left\{ \bigcup_{i=1}^{n-2\lceil \lg n\rceil+1} A_{i,2\lceil \lg n\rceil} \right\} &\le \sum_{i=1}^{n} 1/n^2 \\ &= 1/n , \end{aligned}$$

dato che, per la disuguaglianza di Boole (6.22), la probabilità dell'unione degli eventi è al più uguale alla somma delle probabilità degli eventi individuali. (Notare che la disuguaglianza di Boole vale anche per eventi come questi che non sono indipendenti).

La probabilità che qualunque sequenza di teste sia lunga almeno $2\lceil \lg n \rceil$ è perciò al più $1/n$; quindi la probabilità che la più lunga sequenza abbia lunghezza minore di $2\lceil \lg n \rceil$ è $1 - 1/n$. Dato che ogni sequenza è lunga al più n, la lunghezza attesa della sequenza più lunga è superiormente limitata da

$$(2\lceil \lg n \rceil)(1 - 1/n) + n(1/n) = O(\lg n) .$$

La probabilità che una sequenza di teste ecceda $r\lceil \lg n \rceil$ lanci diminuisce rapidamente con r. Per $r \ge 1$, la probabilità che una sequenza di $r\lceil \lg n \rceil$ teste cominci dalla posizione i è

$$\begin{aligned} \Pr\{A_{i,r\lceil \lg n\rceil}\} &= 1/2^{r\lceil \lg n\rceil} \\ &\le 1/n^r . \end{aligned}$$

Quindi la probabilità che la sequenza più lunga sia almeno $r\lceil \lg n \rceil$, è al più $n/n^r = 1/n^{r-1}$ o, equivalentemente, la probabilità che la sequenza più lunga abbia una lunghezza minore di $r\lceil \lg n \rceil$, è almeno $1 - 1/n^{r-1}$.

Per esempio, per $n = 1000$ lanci, la probabilità di avere una sequenza di almeno $2\lceil \lg n \rceil = 20$ teste è al più $1/n = 1/1000$. La probabilità di avere una sequenza più lunga di $3\lceil \lg n \rceil = 30$ è al più $1/n^2 = 1/1000000$.

Adesso si proverà un limite inferiore complementare: la lunghezza attesa della sequenza di teste più lunga per n lanci della moneta è $\Omega(\lg n)$. Per dimostrare questo limite si cercano sequenze di lunghezza $\lfloor \lg n \rfloor/2$. Dall'equazione (6.47), si ha

$$\begin{aligned} \Pr\{A_{i,\lfloor \lg n\rfloor/2}\} &= 1/2^{\lfloor \lg n\rfloor/2} \\ &\ge 1/\sqrt{n} . \end{aligned}$$

Perciò, la probabilità che una sequenza di almeno $\lfloor \lg n \rfloor/2$ teste non cominci dalla posizione i è al più $1 - 1/\sqrt{n}$. Si possono partizionare gli n lanci della moneta in almeno $\lfloor 2n/\lfloor \lg n \rfloor \rfloor$ gruppi di $\lfloor \lg n \rfloor/2$ lanci consecutivi. Poiché questi gruppi sono formati da lanci indipendenti e mutuamente esclusivi, la probabilità che ciascun gruppo *non sia* una sequenza di lunghezza $\lfloor \lg n \rfloor/2$ è

$$\begin{aligned} (1 - 1/\sqrt{n})^{\lfloor 2n/\lfloor \lg n\rfloor \rfloor} &\le (1 - 1/\sqrt{n})^{2n/\lg n - 1} \\ &\le e^{-(2n/\lg n - 1)/\sqrt{n}} \\ &= O(e^{-\lg n}) \\ &= O(1/n). \end{aligned}$$

Per arrivare a ciò, si è usata la disuguaglianza (2.7), $1 + x \le e^x$.

Quindi, dato che la probabilità che la sequenza più lunga ecceda $\lfloor \lg n \rfloor/2$ è almeno $1 - O(1/n)$ e dato che la sequenza sarà lunga almeno 0, la sua lunghezza attesa è almeno

$$(\lfloor \lg n \rfloor/2)(1 - O(1/n)) + 0 \cdot (1/n) = \Omega(\lg n) .$$

Esercizi

6.6-1 Si supponga di avere lanciato le palline in b contenitori. Ciascun lancio è indipendente ed è equamente probabile che ciascuna pallina finisca in un qualsiasi contenitore. Qual è il numero atteso di lanci che si devono fare prima che almeno un contenitore contenga due palline?

* *6.6-2* Per l'analisi del paradosso del compleanno, è importante che i compleanni siano mutuamente indipendenti o è sufficiente che lo siano a coppie? Motivare la risposta.

* *6.6-3* Quante persone dovrebbero essere invitate ad una festa perché sia probabile che ci siano *tre* persone con lo stesso compleanno?

* *6.6-4* Qual è la probabilità che una k-stringa su un insieme di cardinalità n sia in realtà una k-permutazione? Come si può correlare questo problema con il paradosso dei compleanni?

* *6.6-5* Si supponga che n palline siano lanciate in n contenitori, dove ciascun lancio è indipendente ed è equamente probabile che la pallina finisca in un qualsiasi contenitore. Qual è il numero atteso di contenitori vuoti? Qual è il numero atteso di contenitori con esattamente una pallina?

* *6.6-6* Raffinare il limite inferiore della lunghezza della sequenza di successi mostrando che in n lanci di una moneta perfetta, la probabilità che nessuna sequenza di teste consecutive sia più lunga di $\lg n - 2\lg\lg n$ è minore di $1/n$.

Problemi

6-1 *Palline e contenitori*

In questo problema si analizza l'effetto di varie ipotesi sul numero di modi di mettere n palline in b contenitori diversi.

a. Supporre che le n palline siano diverse e che il loro ordine dentro un contenitore non sia importante. Dedurre che il numero di modi di mettere le palline nei contenitori è b^n.

b. Supporre che le palline siano diverse e, in ogni contenitore, ordinate. Provare che il numero di modi di mettere le palline nei contenitori è $(b + n - 1)!/(b - 1)!$. (*Suggerimento*: si consideri il numero di modi di disporre n palline distinte e $b - 1$ barrette indistinguibili in sequenza).

c. Supporre che le palline siano identiche e quindi che il loro ordine dentro un contenitore non sia importante. Mostrare che il numero di modi di mettere le palline nei contenitori è $\binom{b+n-1}{n}$. (*Suggerimento*: quante delle configurazioni della parte (b) sono ripetute se le palline sono identiche?)

d. Supporre che le palline siano identiche e che nessun contenitore contenga più di una pallina. Mostrare che il numero di modi di mettere le palline è $\binom{b}{n}$.

e. Supporre che le palline siano identiche e che nessun contenitore sia lasciato vuoto. Mostrare che il numero di modi di mettere le palline è $\binom{n-1}{b-1}$.

6-2 *Analisi del programma max*

Il seguente programma determina il valore massimo presente in un array non ordinato $A[1 \ldots n]$.

```
1  max ← −∞
2  for i ← 1 to n
3        do ▷ Confronta A[i] con max.
4           if A[i] > max
5                 then max ← A[i]
```

Si vuole determinare il numero medio di volte in cui viene eseguito l'assegnamento della linea 5. Si assuma che i numeri in A costituiscano una permutazione casuale di n numeri distinti.

a. Se un numero x è scelto casualmente da un insieme di k numeri distinti, qual è la probabilità che x sia il numero più grande dell'insieme?

b. Quando viene eseguita la linea 5 del programma, qual è la relazione tra $A[i]$ e $A[j]$ per $1 \le j \le i$?

c. Per ogni i nell'intervallo $1 \le i \le n$, qual è la probabilità che venga eseguita la linea 5?

d. Siano $s_1, s_2, \ldots, s_n$ n variabili casuali, dove s_i rappresenta il numero di volte (0 o 1) che la linea 5 viene eseguita durante l'iterazione i-esima del ciclo **for**. Qual è il valore atteso $E[s_i]$?

e. Sia $s = s_1 + s_2 + \ldots + s_n$ il numero totale di volte che la linea 5 viene eseguita durante qualche esecuzione del programma. Mostrare che $E[s] = \Theta(\lg n)$.

6-3 *Problema dell'assunzione*

La professoressa Montalpini ha bisogno di assumere un nuovo assistente. Ha fissato i colloqui con n aspiranti e vorrebbe fare la sua scelta soltanto in base alle loro capacità. Purtroppo, le regole dell'università richiedono che dopo ogni colloquio ella immediatamente scarti l'aspirante o gli offra il posto.

La professoressa Montalpini decide di adottare la strategia di selezionare un intero positivo $k < n$, scartando dopo il colloquio i primi k aspiranti e assumendo il primo aspirante successivo che sia meglio qualificato di tutti gli aspiranti precedenti. Se l'aspirante meglio qualificato è tra i primi k, allora scarterà tutti gli n aspiranti. Mostrare che la professoressa Montalpini massimizza le sue possibilità di assumere l'aspirante più qualificato scegliendo k approssimativamente uguale a n/e e che le sue possibilità di assumere l'aspirante più qualificato sono quindi approssimativamente $1/e$.

6-4 *Il contatore probabilistico*

Con un contatore di t bit normalmente si può contare solo fino a $2^t - 1$. Con il ***contatore probabilistico*** di R. Morris, si può contare fino a un valore molto più grande a scapito di un po' di precisione.

Un valore i del contatore rappresenta un conteggio di n_i per $i = 0, 1, \ldots, 2^t - 1$, dove gli n_i formano una sequenza crescente di valori non negativi. Si assuma che il valore iniziale del contatore sia 0, che quindi rappresenta il calcolo di $n_0 = 0$. L'operazione INCREMENT opera in modo probabilistico sul valore i del contatore. Se $i = 2^t - 1$, allora si segnala un errore di overflow, altrimenti il contatore viene aumentato di 1 con probabilità $1/(n_{i+1} - n_i)$, oppure viene lasciato così com'è con probabilità $1 - 1/(n_{i+1} - n_i)$.

Se si seleziona $n_i = i$ per ogni $i \ge 0$, allora il contatore è un contatore normale. Situazioni più interessanti si hanno se si seleziona $n_i = 2^{i-1}$ per $i > 0$ o $n_i = F_i$ (l'i-esimo numero di Fibonacci: si veda il paragrafo 2.2).

Per questo problema, si assuma che n_{2^t-1} sia sufficientemente grande da rendere trascurabile la probabilità di un errore di overflow.

a. Mostrare che il valore atteso rappresentato dal contatore dopo n operazioni INCREMENT è esattamente n.

b. L'analisi della varianza del conteggio rappresentato dal contatore dipende dalla sequenza degli n_i. Si consideri un caso semplice: $n_i = 100i$ per ogni $i \ge 0$. Dare una stima della varianza del valore rappresentato dal registro dopo che sono state eseguite n operazioni INCREMENT.

Note al capitolo

I primi metodi generali per risolvere problemi di probabilità furono discussi in una famosa corrispondenza tra B. Pascal e P. de Fermat, che cominciò nel 1654, nonché in un libro di C. Huygens del 1657. La teoria rigorosa della probabilità cominciò con il lavoro di J. Bernoulli

nel 1713 e di A. De Moivre nel 1730. Ulteriori sviluppi della teoria sono dovuti a P.S. de Laplace, S.-D. Poisson e C.F. Gauss.

Le somme di variabili casuali furono studiate in origine da P.L. Chebyshev e A.A. Markov. La teoria delle probabilità fu assiomatizzata da A.N. Kolmogorov nel 1933. I limiti sulle code di distribuzione furono dati da Chernoff [40] e Hoeffding [99]. Molti lavori pionieristici sulle strutture combinatorie casuali sono dovuti a P. Erdös.

Knuth [121] e Liu [140] sono buoni riferimenti per il calcolo combinatorio elementare. Libri di testo standard come Billingsley [28], Chung [41], Drake [57], Feller [66] e Rozanov [171] offrono un'ampia introduzione al calcolo delle probabilità. Bollobás [30], Hofri [100] e Spencer [179] contengono un'abbondante raccolta di tecniche probabilistiche evolute.

II Ordinamento e selezione

Introduzione

Questa parte contiene diversi algoritmi che risolvono il seguente ***problema di ordinamento***:

Input: una sequenza di n numeri $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: una permutazione (riordinamento) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ della sequenza di input tale che $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

La sequenza di input è generalmente un array di n elementi, benché possa essere rappresentata in qualche altro modo, per esempio come una lista concatenata.

La struttura dei dati

In pratica, i numeri da ordinare raramente rappresentano singoli valori: spesso ciascuno di essi fa parte di una collezione di dati chiamata ***record***. Ciascun record contiene una ***chiave***, che è il valore da ordinare, mentre la parte rimanente del record è costituita dai ***dati satellite***, che sono generalmente collegati alla chiave; in pratica, quando un algoritmo di ordinamento permuta le chiavi, deve eseguire la stessa permutazione anche sui dati satellite. Se ciascun record ha una grande quantità di dati satellite, spesso si permuta un array di puntatori ai record piuttosto che i record stessi in modo da minimizzare il movimento dei dati.

In un certo senso, sono questi dettagli implementativi che distinguono un algoritmo da un corrispondente programma completo. Ordinare singoli numeri o grandi record che contengono numeri è irrilevante rispetto al *metodo* con cui una procedura di ordinamento determina la posizione finale degli elementi. Di conseguenza, quando ci si interessa più strettamente al problema dell'ordinamento si assume che l'input sia costituito soltanto da numeri. La trasformazione di un algoritmo per ordinare numeri in un programma per ordinare record è concettualmente semplice, sebbene in una data situazione di ingegnerizzazione del programma ci potrebbero essere altri aspetti delicati che rendono complessa la programmazione vera e propria.

Algoritmi di ordinamento

Nel Capitolo 1 sono stati presentati due algoritmi che ordinano n numeri reali. Insertion sort impiega un tempo $\Theta(n^2)$ nel caso peggiore. Tuttavia, dato che i suoi cicli annidati sono compatti, esso risulta un algoritmo di ordinamento in loco veloce quando l'input è piccolo. (Si ricordi che un algoritmo di ordinamento ordina ***in loco*** se soltanto un numero costante di elementi dell'array di input è memorizzato fuori dall'array). Il merge sort ha un miglior tempo di esecuzione asintotico, $\Theta(n \lg n)$, ma la procedura MERGE che esso usa non opera in loco.

In questa parte, saranno presentati altri due algoritmi che ordinano numeri reali arbitrari. Heapsort, presentato nel Capitolo 7, ordina n numeri in loco in tempo $O(n \lg n)$; esso usa un'importante struttura di dati, chiamata heap, per realizzare una coda con priorità.

Anche il quicksort, nel Capitolo 8, ordina n numeri in loco, ma il suo tempo di esecuzione nel caso peggiore è $\Theta(n^2)$. Il suo tempo di esecuzione nel caso medio è però $\Theta(n \lg n)$ e generalmente nella pratica è migliore dello heapsort. Come l'insertion sort, il quicksort ha un codice compatto, per cui il fattore costante nascosto del tempo di esecuzione è piccolo: è un algoritmo molto popolare per ordinare grandi array di input.

Insertion sort, merge sort, heapsort e quicksort sono tutti ordinamenti che operano per confronto: determinano la posizione finale degli elementi di un array di input confrontandoli l'uno con l'altro. Il Capitolo 9 comincia introducendo il modello dell'albero di decisione che consente di studiare le limitazioni delle prestazioni degli ordinamenti che operano per confronto. Usando questo modello si dimostra un limite inferiore $\Omega(n \lg n)$ sul tempo di esecuzione nel caso peggiore di qualsiasi ordinamento per confronto di n input, mostrando così che heapsort e merge sort sono ordinamenti per confronto asintoticamente ottimi.

Il Capitolo 9 continua mostrando che si può migliorare questo limite inferiore $\Omega(n \lg n)$ se si possono raccogliere informazioni per ordinare l'input tramite qualcosa di diverso dal confronto tra elementi. L'algoritmo counting sort, per esempio, fa l'ipotesi che i numeri dell'input appartengano all'insieme $\{1, 2, \ldots, k\}$; usando array indicizzati, può ordinare n numeri in tempo $O(k + n)$. Per cui, quando $k = O(n)$, l'algoritmo di counting sort viene eseguito in tempo lineare rispetto alla dimensione dell'array di input. Un algoritmo correlato, radix sort, può essere usato per estendere l'intervallo di applicazione del counting sort: se vi sono n interi da ordinare e ciascun intero ha d cifre e ciascuna cifra è nell'insieme $\{1, 2, \ldots, k\}$, il radix sort può ordinare i numeri in tempo $O(d(n + k))$. Quando d è una costante e k è $O(n)$ radix sort viene eseguito in tempo lineare. Un terzo algoritmo, bucket sort, richiede conoscenze sulla distribuzione probabilistica dei numeri nell'array di input; esso può ordinare n numeri reali uniformemente distribuiti in un intervallo semiaperto $[0,1)$ in tempo $O(n)$ nel caso medio.

Selezione

L'i-esimo numero più piccolo di un insieme di n numeri può essere selezionato, in modo ovvio, ordinando l'input ed accedendo all'i-esimo elemento dell'output. Senza alcuna ipotesi sulla distribuzione dell'input, questo metodo viene eseguito in tempo $\Omega(n \lg n)$, come dimostra il limite inferiore provato nel Capitolo 9.

Nel Capitolo 10 si mostra che si può trovare l'i-esimo elemento più piccolo in tempo $O(n)$, anche quando gli elementi siano arbitrari numeri reali. Si presenta un algoritmo con uno pseudocodice compatto che viene eseguito in tempo $O(n^2)$ nel caso peggiore, ma in tempo lineare nel caso medio. Si fornisce anche un algoritmo più complicato che viene eseguito nel caso peggiore in tempo $O(n)$.

Prerequisiti

In genere questa parte non presenta difficoltà matematiche, ma alcuni paragrafi richiedono concetti matematici più sofisticati. In particolare, le analisi del caso medio di quicksort, bucket sort e dell'algoritmo di selezione usano la probabilità, che è stata già ripresa nel Capitolo 6. L'analisi dell'algoritmo di selezione che richiede tempo lineare nel caso peggiore, coinvolge concetti matematici un po' più complessi rispetto alle altre analisi del caso peggiore presentate in questa parte.

Heapsort

7

In questo capitolo, si introdurrà un altro algoritmo di ordinamento. Come il merge sort, e diversamente dall'insertion sort, il tempo di esecuzione dell'heapsort è $O(n \lg n)$. Come l'insertion sort, e diversamente dal merge sort, l'heapsort ordina in loco: in ogni istante solo un numero costante di elementi dell'array è memorizzato fuori dall'array di input. Quindi l'heapsort presenta i vantaggi dei due algoritmi di ordinamento già discussi.

Inoltre l'heapsort introduce un'altra tecnica di progetto di algoritmi: l'uso di una struttura di dati, denominata "heap", per gestire le informazioni durante l'esecuzione dell'algoritmo. La struttura di dati heap è utile non solo per l'heapsort, ma anche per realizzare efficientemente una coda con priorità. Questa struttura di dati sarà riutilizzata in alcuni algoritmi di capitoli successivi.

Si noti che il termine "heap" originariamente fu coniato nel contesto dell'heapsort, ma in seguito è stato usato per riferirsi alla memoria gestita con il "garbage-collector" come quella fornita dal linguaggio di programmazione Lisp. La struttura di dati che viene presentata *non* è questo tipo di memoria, e dovunque ci sia un riferimento allo heap in questo libro, esso si rifà alla struttura di dati definita in questo capitolo.

7.1 Heap

La struttura di dati ***heap (binario)*** è un array che può essere visto come un albero binario quasi completo (si veda il paragrafo 5.5.3), come mostrato nella figura 7.1. Ogni nodo dell'albero corrisponde a un elemento dell'array che contiene il valore del nodo. L'albero è riempito completamente su tutti i livelli tranne, eventualmente, il più basso che è riempito da sinistra in poi. Un array A che rappresenta uno heap è un oggetto con due attributi: *length*[A], che è il numero di elementi dell'array e *heap-size*[A], il numero di elementi dello heap memorizzati nell'array A. Cioè, benché $A[1 \ldots length[A]]$ possa contenere elementi validi, nessun elemento dopo $A[heap\text{-}size[A]]$, dove $heap\text{-}size[A] \leq length[A]$, è un elemento dello heap. La radice dell'albero è $A[1]$, e se i è l'indice di un nodo, l'indice del padre PARENT(i), del figlio sinistro LEFT(i) e del figlio destro RIGHT(i) possono essere calcolati semplicemente:

```
PARENT(i)
  return ⌊i/2⌋
LEFT(i)
  return 2i
```

Figura 7.1 *Uno heap visto come un albero binario* (a) *e un array* (b). *Il numero dentro ogni nodo è il valore memorizzato in quel nodo. Il numero vicino al nodo è il corrispondente indice nell'array.*

```
RIGHT(i)
  return 2i + 1
```

In molti calcolatori, la procedura LEFT può calcolare $2i$ con una sola istruzione semplicemente traslando di una posizione a sinistra la rappresentazione binaria di i. Analogamente, la procedura RIGHT può velocemente calcolare $2i + 1$ tramite la traslazione della rappresentazione binaria di i di una posizione a sinistra e inserendo 1 nel bit di ordine più basso. La procedura PARENT può calcolare $\lfloor i/2 \rfloor$ tramite la traslazione di i di una posizione verso destra. In una buona realizzazione dello heapsort, queste tre procedure sono spesso realizzate come procedure "macro" o "in-linea".

Gli heap soddisfano anche la ***proprietà di ordinamento parziale dello heap*** (che nel seguito sarà chiamata più semplicemente ***proprietà dello heap***): per ogni nodo i diverso dalla radice,

$$A[\text{PARENT}(i)] \geq A[i], \tag{7.1}$$

cioè il valore di un nodo è minore o uguale al valore del padre. Quindi l'elemento più grande nello heap è memorizzato nella radice e i sottoalberi di qualunque nodo contengono valori non maggiori del valore del nodo stesso.

Si definisce ***altezza*** di un nodo in un albero il numero di archi sul più lungo cammino semplice discendente che va dal nodo a una foglia, e si definisce altezza dell'albero l'altezza della sua radice. Poiché uno heap di n elementi si basa su un albero binario completo, la sua altezza è $\Theta(\lg n)$ (si veda l'Esercizio 7.1-2). Si vedrà che le operazioni di base sugli heap hanno un tempo di esecuzione al più proporzionale all'altezza dell'albero e quindi impiegano un tempo dell'ordine di $O(\lg n)$. Il resto di questo capitolo presenta cinque procedure di base e mostra come usarle in un algoritmo di ordinamento e in una struttura di dati chiamata coda con priorità.

- La procedura HEAPIFY, che ha tempo di esecuzione $O(\lg n)$, è la procedura chiave per mantenere la proprietà dello heap (7.1).
- La procedura BUILD-HEAP, che ha tempo di esecuzione lineare, produce uno heap a partire da un array di input non ordinato.
- La procedura HEAPSORT, che ha tempo di esecuzione $O(n \lg n)$, ordina un array in loco.
- Le procedure EXTRACT-MAX e INSERT, che hanno tempo di esecuzione $O(\lg n)$, permettono di usare la struttura di dati heap come coda con priorità.

Esercizi

7.1-1 Quali sono il numero massimo e il numero minimo di elementi in uno heap di altezza h.

7.1-2 Mostrare che uno heap di n elementi ha altezza $\lfloor \lg n \rfloor$.

7.1-3 Mostrare che l'elemento più grande in un sottoalbero di uno heap è nella radice del sottoalbero.

7.1-4 In uno heap, dove potrebbe risiedere l'elemento più piccolo, assumendo che siano tutti distinti?

7.1-5 Un array ordinato in ordine inverso è uno heap?

7.1-6 La sequenza ⟨23, 17, 14, 6, 13, 10, 1, 5, 7, 12⟩ è uno heap?

7.2 Mantenimento della proprietà dello heap

HEAPIFY è un importante sottoprogramma per gestire gli heap. I suoi input sono un array A e un indice i nell'array. Quando HEAPIFY viene chiamata, si assume che gli alberi binari con radice in LEFT(i) e RIGHT(i) siano heap, ma che $A[i]$ possa essere più piccolo dei suoi figli, violando così la proprietà dello heap (7.1). La funzione di HEAPIFY è di lasciar "scendere" il valore di $A[i]$ nello heap così che il sottoalbero con radice di indice i diventi uno heap.

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  if l ≤ heap-size[A] e A[l] > A[i]
      then largest ← l
      else largest ← i
  if r ≤ heap-size[A] e A[r] > A [largest]
      then largest ← r
  if largest ≠ i
      then scambia A[i] ↔ A[largest]
           HEAPIFY(A, largest)
```

La figura 7.2 illustra il comportamento di HEAPIFY. Ad ogni passo, è determinato il più grande tra $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$ e il suo indice è memorizzato in *largest*. Se $A[i]$ è più grande, allora il sottoalbero con radice nel nodo i è uno heap e la procedura termina. Altrimenti, uno dei due figli ha un elemento più grande, e scambiando $A[i]$ con $A[largest]$, il nodo i e i suoi figli soddisfano la proprietà dello heap. Il nodo *largest*, però, ora ha il valore

Figura 7.2 *L'azione di* HEAPIFY(*A*, 2), *dove heap-size*[*A*] = 10. (**a**) *La configurazione iniziale dello heap viola la proprietà dello heap poiché A*[2] *nel nodo i* = 2 *non è più grande dei suoi figli. La proprietà dello heap è ripristinata per il nodo* 2 *in* (**b**) *scambiando A*[2] *con A*[4], *ma ciò porta a violare la proprietà dello heap per il nodo* 4. *La chiamata ricorsiva* HEAPIFY(*A*, 4) *pone i* = 4. *Dopo lo scambio di A*[4] *con A*[9], *come mostrato in* (**c**), *il nodo* 4 *è sistemato e la chiamata ricorsiva* HEAPIFY(*A*, 9) *non richiede ulteriori cambiamenti della struttura di dati.*

originale di $A[i]$, e quindi il sottoalbero con radice nel nodo *largest* potrebbe violare la proprietà dello heap. Di conseguenza, HEAPIFY deve essere chiamata ricorsivamente su questo sottoalbero.

Il tempo di esecuzione di HEAPIFY su un sottoalbero di dimensione n con radice in un dato nodo i è uguale a $\Theta(1)$, per stabilire la relazione tra gli elementi $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$, più il tempo per far eseguire HEAPIFY su un sottoalbero con radice in uno dei due figli del nodo i. I sottoalberi dei figli hanno ciascuno dimensione al più $2n/3$ – il caso peggiore si verifica quando l'ultimo livello dell'albero è pieno esattamente a metà – e il tempo di esecuzione di HEAPIFY può allora essere descritto dalla ricorrenza

$$T(n) \le T(2n/3) + \Theta(1) .$$

La soluzione della ricorrenza, per il caso 2 del teorema principale (Teorema 4.1), è $T(n) = O(\lg n)$. In alternativa, si può definire come $O(h)$ il tempo di esecuzione di HEAPIFY su un nodo di altezza h.

Esercizi

7.2-1 Usando la figura 7.2 come modello, illustrare le operazioni di HEAPIFY(*A*, 3) sull'array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

7.2-2 Qual è l'effetto di chiamare HEAPIFY(*A*, *i*) se l'elemento $A[i]$ è più grande dei suoi figli?

7.2-3 Qual è l'effetto di chiamare HEAPIFY(*A*, *i*) per $i > \textit{heap-size}[A]/2$?

7.2-4 Il codice di HEAPIFY è abbastanza efficiente in termini di fattori costanti, tranne che per la chiamata ricorsiva nella linea 10, che potrebbe provocare in qualche compilatore la produzione di codice inefficiente. Scrivere una procedura HEAPIFY efficiente che usi un costrutto di controllo iterativo (un ciclo) invece della ricorsione.

7.2-5 Mostrare che, nel caso peggiore, il tempo di esecuzione di HEAPIFY su uno heap di dimensione n è $\Omega(\lg n)$. (*Suggerimento*: per uno heap con n nodi, assegnare valori ai nodi in modo che HEAPIFY sia chiamata ricorsivamente su ogni nodo del cammino dalla radice fino a una foglia).

7.3 Costruzione di uno heap

Si può usare la procedura HEAPIFY in modo bottom-up per convertire un array $A[1 \ldots n]$, con $n = \textit{length}[A]$, in uno heap.

Per cominciare, poiché gli elementi nel sottoarray $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ sono tutte foglie dell'albero, ognuno di essi è uno heap di un solo elemento. La procedura BUILD-HEAP attraversa i restanti nodi dell'albero ed esegue HEAPIFY su ognuno di essi. L'ordine in cui i nodi sono trattati garantisce che i sottoalberi con radice nei figli di un nodo i sono heap, prima che HEAPIFY venga eseguita su quel nodo.

```
BUILD-HEAP(A)
  heap-size[A] ← length[A]
  for i ← ⌊length[A]/2⌋ downto 1
      do HEAPIFY(A, i)
```

La figura 7.3 mostra un esempio dell'esecuzione di BUILD-HEAP.

Si può calcolare un semplice limite superiore sul tempo di esecuzione di BUILD-HEAP nel modo seguente. Ogni chiamata a HEAPIFY richiede un tempo $O(\lg n)$ e tali chiamate sono $O(n)$. Quindi il tempo di esecuzione è al più $O(n \lg n)$. Questo limite superiore, sebbene corretto, non è asintoticamente stretto.

Si può derivare un limite più stretto osservando che il tempo per far eseguire HEAPIFY su un nodo varia con l'altezza del nodo nell'albero e le altezze di molti nodi sono piccole. Un'analisi più rigorosa si basa sulla proprietà che uno heap di n elementi ha altezza $\lfloor \lg n \rfloor$ (si veda l'Esercizio 7.1-2) e al più $\lceil n/2^{h+1} \rceil$ nodi di altezza h (si veda l'Esercizio 7.3-3).

Il tempo richiesto da HEAPIFY quando è chiamata su un nodo di altezza h è $O(h)$, quindi si può esprimere il costo totale di BUILD-HEAP come:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) . \qquad (7.2)$$

Figura 7.3 *L'esecuzione di* BUILD-HEAP, *mostrata attraverso la struttura di dati prima della chiamata a* HEAPIFY *nella linea* 3 *di* BUILD-HEAP. (a) *Un array di input A di* 10 *elementi e l'albero binario che lo rappresenta. La figura mostra che l'indice i del ciclo punta al nodo* 5 *prima della chiamata a* HEAPIFY(*A*, *i*). (b) *La struttura di dati risultante. L'indice i del ciclo per l'iterazione successiva punta al nodo* 4. (c)-(e) *Le iterazioni successive del ciclo* **for** *nella* BUILD-HEAP. *Si osservi che se* HEAPIFY *viene chiamata su un nodo, i due sottoalberi di quel nodo sono heap entrambi.* (f) *Lo heap dopo che* BUILD-HEAP *è terminata.*

L'ultima sommatoria può essere valutata sostituendo $x = 1/2$ nella formula (3.6), per cui vale

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 .$$

Quindi il tempo di esecuzione di BUILD-HEAP può essere così limitato:

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n) .$$

In conclusione, si può costruire uno heap da un array non ordinato in tempo lineare.

Esercizi

7.3-1 Usando la figura 7.3 come modello, illustrare le operazioni di BUILD-HEAP sull'array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

7.3-2 Perché si fa decrescere l'indice i del ciclo nella linea 2 di BUILD-HEAP da $\lfloor length[A]/2 \rfloor$ a 1 invece di farlo crescere da 1 a $\lfloor length[A]/2 \rfloor$?

* *7.3-3* Mostrare che in qualunque heap di n elementi vi sono al più $\lceil n/2^{h+1} \rceil$ nodi di altezza h.

7.4 L'algoritmo heapsort

L'algoritmo heapsort comincia con il costruire uno heap sull'array di input $A[1 \ldots n]$, con $n = length[A]$, usando BUILD-HEAP. Poiché l'elemento più grande dell'array è memorizzato nella radice $A[1]$, esso può essere posto nella sua corretta posizione finale scambiandolo con $A[n]$. Se si "cancella" il nodo n dallo heap (decrementando *heap-size*[A]), si osserva che $A[1 \ldots (n-1)]$ può facilmente essere trasformato in uno heap. I figli della radice rimangono heap, ma il nuovo elemento nella radice può violare la proprietà (7.1) dello heap. Per ripristinare la proprietà dello heap, è comunque sufficiente chiamare HEAPIFY(A, 1), che rende via via $A[1 \ldots (n-1)]$ uno heap. L'algoritmo heapsort ripete questo processo per lo heap di dimensione $n-1$ fino allo heap di dimensione 2.

```
HEAPSORT(A)
1   BUILD-HEAP(A)
2   for i ← length[A] downto 2
3       do scambia A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          HEAPIFY(A, 1)
```

La figura 7.4 mostra un esempio dell'operazione di heapsort dopo che lo heap è stato costruito per la prima volta. Ogni heap è mostrato all'inizio di un'iterazione del ciclo **for** della linea 2.

La procedura HEAPSORT impiega un tempo $O(n \lg n)$, poiché la chiamata a BUILD-HEAP impiega un tempo $O(n)$ e ognuna delle $n-1$ chiamate a HEAPIFY impiega un tempo $O(\lg n)$.

Figura 7.4 *L'esecuzione di* HEAPSORT. (a) *La struttura di dati heap subito dopo che è stata costruita con* BUILD-HEAP. (b-j) *Lo heap subito dopo ogni chiamata di* HEAPIFY *nella riga 5. È evidenziato il valore di i in quell'istante. Nello heap rimangono solo i nodi chiari.* (k) *Il risultante array A ordinato.*

Esercizi

7.4-1 Usando la figura 7.4 come modello, illustrare l'operazione di HEAPSORT sull'array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

7.4-2 Qual è il tempo di esecuzione di heapsort su un array A di lunghezza n già ordinato in ordine crescente? E se l'ordine è decrescente?

7.4-3 Mostrare che il tempo di esecuzione di heapsort è $\Omega(n \lg n)$ nel caso peggiore.

7.5 Code con priorità

Heapsort è un buon algoritmo, ma una buona realizzazione del quicksort, presentato nel Capitolo 8, di solito è migliore in pratica. Ciò nonostante, la struttura di dati heap ha un'utilità enorme. In questo paragrafo, sarà presentata una delle più conosciute applicazioni dello heap: il suo uso come efficiente coda con priorità.

Una ***coda con priorità*** è una struttura di dati per mantenere un insieme S di elementi, ognuno con un valore associato chiamato ***chiave***. Una coda con priorità offre le seguenti operazioni.

INSERT(S, x) inserisce l'elemento x nell'insieme S. Quest'operazione potrebbe essere scritta come $S \leftarrow S \cup \{x\}$.

MAXIMUM(S) restituisce l'elemento di S con chiave più grande.

EXTRACT-MAX(S) rimuove e restituisce l'elemento di S con chiave più grande.

Una applicazione delle code con priorità è l'allocazione di processi su un calcolatore condiviso. La coda con priorità mantiene traccia dei processi che devono essere eseguiti e delle loro priorità relative. Quando un processo termina o è interrotto, il processo con priorità più alta è selezionato tra quelli in attesa usando EXTRACT-MAX. Un nuovo processo può essere aggiunto alla coda in qualunque momento usando INSERT.

Una coda con priorità può essere usata nella simulazione di un sistema guidato dagli eventi. Gli elementi nella coda sono eventi che devono essere simulati, ognuno con associato il tempo in cui deve avvenire, che serve come chiave. Gli eventi devono essere simulati seguendo come ordine il loro tempo di occorrenza, perché la simulazione di un evento può provocare la simulazione di altri eventi nel futuro. Per quest'applicazione, è naturale invertire l'ordine lineare della coda con priorità e offrire le operazioni MINIMUM e EXTRACT-MIN invece di MAXIMUM e EXTRACT-MAX. Il programma di simulazione usa EXTRACT-MIN ad ogni passo e sceglie il prossimo evento da simulare. Via via che nuovi eventi si presentano, essi sono inseriti nella coda con priorità usando INSERT.

Non sorprende che per realizzare una coda con priorità si possa usare uno heap. L'operazione HEAP-MAXIMUM restituisce l'elemento più grande dello heap in tempo $\Theta(1)$ restituendo semplicemente il valore $A[1]$ dello heap. La procedura HEAP-EXTRACT-MAX è simile al corpo del ciclo **for** (linee 3-5) della procedura HEAPSORT:

```
HEAP-EXTRACT-MAX(A)
  if heap-size[A] < 1
    then error "heap underflow"
  max ← A[1]
  A[1] ← A[heap-size[A]]
  heap-size[A] ← heap-size[A] - 1
  HEAPIFY(A, 1)
  return max
```

Il tempo di esecuzione di HEAP-EXTRACT-MAX è $O(\lg n)$, poiché esegue soltanto una quantità costante di lavoro oltre al tempo $O(\lg n)$ di HEAPIFY.

La procedura HEAP-INSERT inserisce un nodo nello heap A. Per farlo, prima espande lo heap aggiungendo una nuova foglia all'albero. Poi, in un modo che ricorda il ciclo di inserimento (linee 5-7) dell'INSERTION-SORT del paragrafo 1.1, percorre un cammino da questa foglia verso la radice per trovare un posto appropriato al nuovo elemento.

```
HEAP-INSERT(A, key)
1   heap-size[A] ← heap-size[A] + 1
2   i ← heap-size[A]
3   while i > 1 e A [PARENT(i)] < key
4       do  A[i] ← A [PARENT(i)]
5           i ← PARENT(i)
6   A[i] ← key
```

La figura 7.5 mostra un esempio di operazione HEAP-INSERT. Il tempo di esecuzione di HEAP-INSERT su uno heap di n elementi è $O(\lg n)$, poiché il cammino seguito dalla nuova foglia alla radice ha lunghezza $O(\lg n)$.

Per riassumere, uno heap può offrire una qualunque operazione di coda con priorità su un insieme di dimensione n in tempo $O(\lg n)$.

Esercizi

7.5-1 Usando la figura 7.5 come modello, illustrare l'operazione di HEAP-INSERT(A, 10) sullo heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

7.5-2 Illustrare inoltre l'operazione di HEAP-EXTRACT-MAX sullo heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

7.5-3 Mostrare come realizzare una coda primo-arrivato-primo-servito (FIFO) con una coda con priorità. Mostrare come realizzare una pila usando una coda con priorità. (Code FIFO e pile sono definite nel paragrafo 11.1).

7.5-4 Realizzare una procedura HEAP-INCREASE-KEY(A, i, k), che ponga $A[i] \leftarrow \max(A[i], k)$ e aggiorni la struttura heap appropriatamente e che richieda $O(\lg n)$ tempo.

7.5-5 L'operazione HEAP-DELETE(A, i) cancella l'elemento del nodo i dallo heap A. Realizzare una procedura HEAP-DELETE che venga eseguita in tempo $O(\lg n)$ su uno heap di n elementi.

7.5-6 Descrivere un algoritmo con tempo di esecuzione $O(n \lg k)$ per fondere k liste ordinate in una lista ordinata, dove n è il numero totale di elementi di tutte le liste in input. (*Suggerimento*: usare uno heap per la fusione delle k liste).

Figura 7.5 *L'esecuzione di* HEAP-INSERT. (a) *Lo heap della figura 7.4* (a) *prima che si inserisca un nodo con chiave* 15. (b) *Una nuova foglia è aggiunta all'albero.* (c) *I valori sul cammino dalla nuova foglia alla radice sono risistemati finché non si trova un posto per la chiave* 15. (d) *La chiave* 15 *è inserita.*

Problemi

7-1 Costruzione di uno heap con l'inserzione

La procedura BUILD-HEAP del paragrafo 7.3 può essere realizzata usando ripetutamente HEAP-INSERT per inserire gli elementi nello heap. Si consideri la seguente realizzazione:

```
BUILD-HEAP'(A)
1   heap-size[A] ← 1
2   for i ← 2 to length[A]
3       do HEAP-INSERT(A, A[i])
```

a. Le procedure BUILD-HEAP e BUILD-HEAP', se sono eseguite sullo stesso array di input, creano sempre lo stesso heap? Dimostrare che lo fanno o fornire un controesempio.

b. Mostrare che, nel caso peggiore, BUILD-HEAP' richiede un tempo $\Theta(n \lg n)$ per costruire uno heap di n elementi.

7-2 ***Analisi di heap d-ari***

Uno ***heap d-ario*** è come uno heap binario, ma i nodi hanno d figli invece di 2.

a. Come si potrebbe rappresentare uno heap d-ario in un array?

b. Qual è l'altezza di uno heap d-ario di n elementi espresso in termini di n e d?

c. Realizzare una procedura efficiente di EXTRACT-MAX. Analizzare il suo tempo di esecuzione in termini di d e n.

d. Realizzare una procedura efficiente di INSERT. Analizzare il suo tempo di esecuzione in termini di d e n.

e. Realizzare una procedura efficiente di HEAP-INCREASE-KEY(A, i, k), che ponga $A[i] \leftarrow \max(A[i], k)$ e aggiorni la struttura heap appropriatamente. Analizzare il suo tempo di esecuzione in termini di d e n.

Note al capitolo

L'algoritmo heapsort fu progettato da Williams [202], che descrisse anche come realizzare una coda con priorità tramite uno heap. La procedura BUILD-HEAP fu proposta da Floyd [69].

Quicksort

8

Il quicksort è un algoritmo di ordinamento il cui tempo di esecuzione nel caso peggiore è $\Theta(n^2)$ su un array di input di n elementi. Malgrado questo tempo di esecuzione lento nel caso peggiore, il quicksort è spesso la migliore scelta pratica di ordinamento perché in media è notevolmente efficiente: il suo tempo medio di esecuzione è $\Theta(n \lg n)$, e i fattori costanti nascosti nella notazione $\Theta(n \lg n)$ sono sufficientemente piccoli. Offre inoltre il vantaggio di ordinare in loco (si veda il paragrafo 1.1) e funziona bene anche in ambienti con memoria virtuale.

Il paragrafo 8.1 descrive l'algoritmo e un importante sottoprogramma usato dal quicksort per il partizionamento. Poiché il comportamento del quicksort è complesso si comincerà nel paragrafo 8.2 con una discussione intuitiva delle sue prestazioni, rimandandone l'analisi formale alla fine del capitolo. Il paragrafo 8.3 presenta due versioni del quicksort che usano un generatore di numeri casuali. Questi algoritmi "randomizzati" presentano molti vantaggi. Il loro tempo di esecuzione nel caso medio è buono, e nessun particolare input provoca il comportamento del caso peggiore. Una delle versioni randomizzate del quicksort è analizzata nel paragrafo 8.4, dove sarà mostrato che il suo tempo di esecuzione è $O(n^2)$ nel caso peggiore e $O(n \lg n)$ nel caso medio.

8.1 Descrizione del quicksort

Il quicksort, come il merge sort, è basato sul paradigma divide-et-impera introdotto nel paragrafo 1.3.1. Il processo divide-et-impera per ordinare un tipico sottoarray $A[p \ldots r]$ richiede le tre fasi qui descritte.

Divide: l'array $A[p \ldots r]$ è ripartito (risistemato) in due sottoarray non vuoti $A[p \ldots q]$ e $A[q+1 \ldots r]$ in modo tale che ogni elemento di $A[p \ldots q]$ sia minore o uguale a qualunque elemento di $A[q+1 \ldots r]$. L'indice q è calcolato da questa procedura di partizionamento.

Impera: i due sottoarray $A[p \ldots q]$ e $A[q+1 \ldots r]$ sono ordinati con chiamate ricorsive a quicksort.

Ricombina: poiché i sottoarray sono ordinati in loco, non è richiesto alcuno sforzo per ricombinarli: l'intero array $A[p \ldots r]$ è subito ordinato. La seguente procedura realizza il quicksort.

```
QUICKSORT (A, p, r)
    if p < r
      then q ← PARTITION (A, p, r)
           QUICKSORT (A, p, q)
           QUICKSORT (A, q + 1, r)
```

Per ordinare l'intero array A, la chiamata iniziale è QUICKSORT $(A, 1, length[A])$.

Partizionamento dell'array

Il cuore dell'algoritmo è la procedura PARTITION, che risistema il sottoarray $A[p \ldots r]$ in loco.

```
PARTITION (A, p, r)
1   x ← A[p]
2   i ← p − 1
3   j ← r + 1
4   while TRUE
5       do repeat j ← j − 1
6              until A[j] ≤ x
7           repeat i ← i + 1
8              until A[i] ≥ x
9          if i < j
10             then scambia A[i] ↔ A[j]
11             else return j
```

La figura 8.1 mostra come funziona PARTITION. Prima viene selezionato un elemento $x = A[p]$ da $A[p \ldots r]$ come elemento "perno" attorno al quale partizionare $A[p \ldots r]$. Quindi vengono fatte crescere le due regioni $A[p \ldots i]$ e $A[j \ldots r]$ rispettivamente dall'inizio e dalla fine di $A[p \ldots r]$ così che ogni elemento in $A[p \ldots i]$ sia minore o uguale a x e ogni elemento in $A[j \ldots r]$ sia maggiore o uguale a x. Inizialmente $i = p - 1$ e $j = r + 1$ e le due regioni sono vuote.

Nel corpo del ciclo while, l'indice j è decrementato e l'indice i è incrementato, nelle linee 5-8, finché $A[i] \geq x \geq A[j]$. Assumendo che queste disuguaglianze siano strette, $A[i]$ è troppo grande per appartenere alla regione bassa e $A[j]$ è troppo piccolo per appartenere alla regione alta. Così, scambiando $A[i]$ e $A[j]$, come è fatto nella linea 10, si possono estendere le due regioni. (Se le disuguaglianze non sono strette, lo scambio può essere eseguito comunque).

Il corpo del ciclo while si ripete finché $i \geq j$; a questo punto l'intero array $A[p \ldots r]$ è stato ripartizionato in due sottoarray $A[p \ldots q]$ e $A[q + 1 \ldots r]$, dove $p \leq q < r$, tali che nessun elemento di $A[p \ldots q]$ è più grande di qualunque elemento di $A[q + 1 \ldots r]$. Il valore $q = j$ è restituito alla fine della procedura.

Concettualmente, la procedura di partizione esegue una funzione semplice: pone gli elementi più piccoli di x nella regione bassa dell'array e quelli più grandi di x nella regione alta. Vi sono aspetti tecnici che rendono però lo pseudocodice di PARTITION un po' complicato. Per esempio gli indici i e j non devono mai riferirsi a elementi del sottoarray $A[p \ldots r]$ fuori dai limiti, ma ciò non è evidente dal codice. Inoltre, è importante che $A[p]$ sia usato come elemento perno. Se invece venisse usato $A[r]$ e se esso fosse anche l'elemento più grande nel sottoarray $A[p \ldots r]$, allora PARTITION restituirebbe a QUICKSORT il valore $q = r$, e QUICKSORT andrebbe in ciclo per sempre. Il Problema 8-1 richiede di provare che PARTITION è corretta.

Il tempo di esecuzione di PARTITION su un array $A[p \ldots r]$ è $\Theta(n)$, dove $n = r - p + 1$ (si veda l'Esercizio 8.1-3).

Figura 8.1 *Esempio di esecuzione di* PARTITION *su un array. Gli elementi grigio chiaro dell'array sono stati messi nella partizione corretta e quelli scuri non sono ancora nella loro partizione.* (a) *L'array di input, con i valori iniziali di i e j fuori dai limiti sinistro e destro dell'array. Si ripartiziona attorno a* $x = A[p] = 5$. (b) *La posizione di i e j nella linea 9 della prima iterazione del ciclo* **while**. (c) *Il risultato dello scambio degli elementi puntati da i e j nella linea 10.* (d) *La posizione di i e j alla riga 9 nella seconda iterazione del ciclo* **while**. (e) *La posizione di i e j alla riga 9 nella terza e ultima iterazione del ciclo* **while**. *La procedura termina perché* $i \geq j$, *ed è restituito il valore* $q = j$. *Gli elementi dell'array che precedono e includono* $A[j]$ *sono minori o uguali a* $x = 5$, *e gli elementi dell'array dopo* $A[j]$ *sono maggiori o uguali a* $x = 5$.

Esercizi

8.1-1 Usando la figura 8.1 come modello, illustrare le operazioni di PARTITION sull'array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$.

8.1-2 Quale valore di q restituisce PARTITION se tutti gli elementi nell'array $A[p \ldots r]$ hanno lo stesso valore?

8.1-3 Spiegare brevemente perché il tempo di esecuzione di PARTITION su un sottoarray di dimensione n è $\Theta(n)$.

8.1-4 Come si potrebbe modificare QUICKSORT per ordinare in ordine decrescente?

8.2 Prestazioni del quicksort

Il tempo di esecuzione di quicksort dipende dal fatto che il partizionamento sia bilanciato o sbilanciato e ciò a sua volta dipende da quali elementi sono usati per il partizionamento. Se il partizionamento è bilanciato, l'algoritmo viene eseguito con la stessa velocità asintotica del merge sort. Se il partizionamento è sbilanciato, però, può essere asintoticamente lento quanto l'insertion sort. In questo paragrafo si studieranno le prestazioni del quicksort in funzione del bilanciamento della partizione.

Figura 8.2 *Un albero di ricorrenza per* QUICKSORT *in cui la procedura* PARTITION *mette sempre solo un elemento su un lato della partizione (il caso peggiore). Il tempo di esecuzione che ne risulta è* $\Theta(n^2)$.

Partizionamento peggiore

Il comportamento peggiore del quicksort avviene quando il sottoprogramma di partizionamento produce una regione con $n-1$ elementi e una con un solo elemento. (Quest'affermazione è dimostrata nel paragrafo 8.4.1).

Si assuma che questo partizionamento sbilanciato avvenga ad ogni passo dell'algoritmo. Poiché il partizionamento richiede tempo $\Theta(n)$ e $T(1) = \Theta(1)$, la ricorrenza per il tempo di esecuzione è

$$T(n) = T(n-1) + \Theta(n) .$$

Per valutare questa ricorrenza, si osservi che $T(1) = \Theta(1)$ e quindi si iteri:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= \sum_{k=1}^{n} \Theta(k) \\ &= \Theta\left(\sum_{k=1}^{n} k\right) \\ &= \Theta(n^2) . \end{aligned}$$

Si ottiene l'ultima riga osservando che $\sum_{k=1}^{n} k$ è la serie aritmetica (3.2). La figura 8.2 mostra un albero di ricorrenza per l'esecuzione di quicksort nel caso peggiore. (Si veda nel paragrafo 4.2 una presentazione degli alberi di ricorrenza).

Di conseguenza, se il partizionamento è il più sbilanciato possibile per ogni passo ricorsivo dell'algoritmo, allora il tempo di esecuzione è $\Theta(n^2)$. Quindi il tempo di esecuzione di quicksort nel caso peggiore non è migliore di quello dell'insertion sort. Inoltre, il tempo di esecuzione $\Theta(n^2)$ si verifica quando l'array di input è già ordinato – una situazione frequente, in cui l'insertion sort è eseguito in tempo $O(n)$.

Figura 8.3 *Un albero di ricorrenza per* QUICKSORT *in cui* PARTITION *bilancia sempre i due lati della partizione in modo uguale (caso migliore). Il tempo di esecuzione che ne risulta è* $\Theta(n \lg n)$.

Partizionamento migliore

Se la procedura di partizionamento produce due regioni di dimensione $n/2$, il quicksort viene eseguito molto più velocemente. La ricorrenza è allora:

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

che, dal caso 2 del teorema principale (Teorema 4.1), ha come soluzione $T(n) = \Theta(n \lg n)$. Pertanto il partizionamento migliore produce un algoritmo molto più veloce. La figura 8.3 mostra l'albero di ricorrenza per l'esecuzione di quicksort nel caso migliore.

Partizionamento bilanciato

Il tempo di esecuzione di quicksort nel caso medio è molto più vicino al caso migliore che al peggiore, come sarà analizzato nel paragrafo 8.4. Per comprendere ciò, bisogna analizzare come il bilanciamento della partizione si rifletta sulla ricorrenza che descrive il tempo di esecuzione.

Si supponga, per esempio, che l'algoritmo di partizionamento produca sempre una suddivisione di 9 a 1, che a prima vista sembra sufficientemente sbilanciata. Allora si ottiene la ricorrenza:

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

per il tempo di esecuzione di quicksort, dove per comodità n è stato sostituito a $\Theta(n)$. La figura 8.4 mostra l'albero di ricorrenza per questa ricorrenza. Si noti che ogni livello dell'albero ha un costo n, finché è raggiunta una condizione al contorno alla profondità $\log_{10} n = \Theta(\lg n)$, e quindi i livelli hanno al più un costo n. La ricorrenza termina alla profondità $\log_{10/9} n = \Theta(\lg n)$. Così, con una suddivisione di 9 a 1 ad ogni livello di ricorrenza, che intuitivamente sembra abbastanza sbilanciata, l'algoritmo quicksort viene eseguito in tempo $\Theta(n \lg n)$: asintoticamente lo stesso che si avrebbe se la suddivisione fosse esattamente a metà. In realtà anche una suddivisione 99 a 1 mantiene un tempo di esecuzione $\Theta(n \lg n)$. Il motivo è che qualunque suddivisione di proporzionalità *costante* produce un albero di ricorrenza di profondità $\Theta(\lg n)$.

Figura 8.4 *Un albero di ricorrenza per* QUICKSORT *in cui* PARTITION *produce sempre una suddivisione 9 a 1, mantenendo un tempo di esecuzione di* $\Theta(n \lg n)$.

in cui il costo di ogni livello è $O(n)$. Il tempo di esecuzione è quindi $\Theta(n \lg n)$ ogni volta che la suddivisione abbia proporzionalità costante.

Concetti intuitivi sul caso medio

Per sviluppare una nozione chiara del caso medio per quicksort, bisogna fare un'ipotesi su quanto frequentemente ci si aspetta di incontrare i vari tipi di input. Un'ipotesi comune è che tutte le permutazioni dei numeri in input siano equamente probabili. Si discuterà quest'ipotesi nel prossimo paragrafo, ma intanto si studieranno subito le sue implicazioni.

Quando si esegue quicksort su un array di input preso a caso, è improbabile che il partizionamento avvenga nello stesso modo ad ogni livello, come è stato ipotizzato nella precedente analisi informale. È ragionevole aspettarsi che qualche suddivisione sia ben bilanciata e che qualcuna non lo sia. Per esempio, l'Esercizio 8.2-5 chiede di mostrare che circa l'80 per cento delle volte PARTITION produce una suddivisione più bilanciata di 9 a 1 e che circa il 20 per cento delle volte produce una suddivisione che è meno bilanciata di 9 a 1.

Nel caso medio, PARTITION produce un misto di suddivisioni "buone" e "cattive". In un albero di ricorrenza dell'esecuzione di PARTITION nel caso medio, le suddivisioni buone e cattive sono distribuite lungo l'albero in modo casuale. Si supponga però per semplicità che le suddivisioni buone e cattive si alternino sui livelli dell'albero e che le buone siano i casi migliori di partizionamento e le cattive siano i casi peggiori. La figura 8.5(a) mostra le suddivisioni su due livelli consecutivi nell'albero di ricorrenza. Nella radice dell'albero, il costo è n per il partizionamento e i sottoarray prodotti hanno dimensione $n - 1$ e 1: il caso peggiore. Al livello successivo, il sottoarray di dimensione $n - 1$ è partizionato in due sottoarray di dimensione $(n - 1)/2$: il caso migliore. Si assuma che il costo della condizione al contorno sia 1 per il sottoarray di dimensione 1.

La combinazione di una suddivisione cattiva seguita da una buona produce tre sottoarray con dimensione: 1, $(n - 1)/2$ e $(n - 1)/2$ con un costo combinato di $2n - 1 = \Theta(n)$. Di certo, questa situazione non è peggiore di quella della figura 8.5(b), in cui una partizione su un

Figura 8.5 (a) *Due livelli di un albero di ricorrenza per l'algoritmo di quicksort. Il partizionamento nella radice costa n e produce una suddivisione "svantaggiosa": due sottoarray di dimensione* 1 *e* $n - 1$. *D'altra parte, il partizionamento del sottoarray di dimensione* $n - 1$ *costa* $n - 1$ *e produce una suddivisione "vantaggiosa": due sottoarray di dimensione* $(n - 1)/2$. (b) *Un singolo livello di un albero di ricorrenza che risulta peggiore dei livelli combinati di (a), malgrado sia molto ben bilanciato.*

singolo livello produce due sottoarray di dimensione $(n - 1)/2 + 1$ e $(n - 1)/2$ con un costo $n = \Theta(n)$. Quest'ultima situazione è molto vicina al bilanciamento, certamente più della suddivisione 9 a 1. Intuitivamente, il costo $\Theta(n)$ della cattiva suddivisione può essere assorbito nel costo $\Theta(n)$ della buona, quindi il costo risultante è buono. Di conseguenza, il tempo di esecuzione di quicksort, quando sui livelli si alternano partizioni buone e cattive, è come il tempo di esecuzione quando si hanno soltanto partizioni buone: ancora $O(n \lg n)$, ma con una costante nascosta dalla notazione O leggermente più grande. Verrà fatta un'analisi rigorosa del caso medio nel paragrafo 8.4.2.

Esercizi

8.2-1 Mostrare che il tempo di esecuzione di QUICKSORT è $\Theta(n \lg n)$ se tutti gli elementi dell'array A hanno lo stesso valore.

8.2-2 Mostrare che il tempo di esecuzione di QUICKSORT è $\Theta(n^2)$ se l'array A contiene elementi distinti ed è ordinato in ordine decrescente.

8.2-3 Le banche spesso registrano le transazioni su un conto seguendo l'ordine in cui sono state eseguite nel tempo, ma molte persone potrebbero preferire ricevere il loro estratto conto bancario con gli assegni ordinati per numero. Le persone di solito scrivono gli assegni ordinati per numero e i commercianti di solito li riscuotono con ragionevole prontezza. Il problema di convertire l'ordinamento per data di transazione in ordinamento per numero di assegno è quindi un tipico problema di ordinamento di input quasi ordinato. Dimostrare che la procedura INSERTION-SORT su questo problema dovrebbe tendere a superare le prestazioni della procedura QUICKSORT.

8.2-4 Si supponga che le suddivisioni di quicksort, ad ogni livello, siano in proporzione α a $1 - \alpha$ dove $0 < \alpha \leq 1/2$ è una costante. Mostrare che la profondità minima di una foglia nell'albero di ricorrenza è pressappoco $-\lg n / \lg \alpha$ e la profondità massima è approssimativamente $-\lg n / \lg(1 - \alpha)$. (Trascurare l'arrotondamento a intero).

* *8.2-5* Dimostrare che, per qualsiasi costante $0 < \alpha \leq 1/2$, la probabilità che PARTITION produca, su un array di input a caso, una partizione più bilanciata del rapporto $1 - \alpha$ ad α è di $1 - 2\alpha$ circa. Per quale valore di α è ugualmente probabile che la partizione risulti più bilanciata di $1 - \alpha$ ad α oppure meno?

8.3 Versione randomizzata del quicksort

Nell'analizzare il comportamento di quicksort nel caso medio, è stata fatta l'ipotesi che tutte le permutazioni dei numeri in input fossero equamente probabili. Quando questa ipotesi sulla distribuzione dell'input è valida, molte persone considerano il quicksort l'algoritmo ideale per input sufficientemente grandi. In situazioni reali, però, non ci si può aspettare che questa ipotesi valga sempre. (Si veda l'Esercizio 8.2-3). Questo paragrafo introduce il concetto di algoritmo randomizzato e presenta due versioni randomizzate di quicksort che superano l'ipotesi che tutte le permutazioni dei numeri in input siano equamente probabili.

Un'alternativa all'*ipotesi* sulla distribuzione di input è di *imporre* una distribuzione. Per esempio, si supponga che, prima di ordinare l'array, quicksort permuti a caso gli elementi per rafforzare la proprietà che ogni permutazione sia equamente probabile. (L'Esercizio 8.3-4 richiede un algoritmo che permuti casualmente gli elementi di un array di dimensione n in tempo $O(n)$.) Questa modifica non migliora il tempo di esecuzione del caso peggiore dell'algoritmo, ma rende il tempo di esecuzione indipendente dall'input.

Un algoritmo si definisce ***randomizzato*** se il suo comportamento è determinato non solo dall'input ma anche da valori prodotti da un ***generatore di numeri casuali***. Si assuma di avere a disposizione un generatore di numeri casuali RANDOM. Una chiamata a RANDOM (a, b) restituisce un intero tra a e b, compresi, essendo equamente probabile la restituzione di uno qualunque di tali interi. Per esempio RANDOM (0,1) restituisce 0 con probabilità 1/2 e 1 con probabilità 1/2. Ogni intero restituito da RANDOM è indipendente dall'intero restituito nella chiamata precedente. Si può immaginare RANDOM come un dado con $(b - a + 1)$ facce che rotola per ottenere il suo output. (In pratica, molti ambienti di programmazione offrono un ***generatore di numeri pseudo-casuali***: un algoritmo deterministico che restituisce numeri che statisticamente "sembrano" casuali.)

Questa versione randomizzata di quicksort ha una proprietà interessante che possiedono anche molti altri algoritmi randomizzati: *nessun input particolare provoca il caso peggiore nel comportamento dell'algoritmo*. Il caso peggiore dipende invece dal generatore di numeri casuali. Anche intenzionalmente, non si riesce a produrre un cattivo array di input, poiché le permutazioni casuali rendono irrilevante l'ordine dell'input. L'algoritmo randomizzato si comporta male solo se il generatore di numeri casuali produce una permutazione sfortunata da ordinare. L'Esercizio 13.4-4 mostra che quasi tutte le permutazioni consentono al quicksort di comportarsi bene quasi come nel caso medio: vi sono *pochissime* permutazioni che causano un comportamento simile a quello del caso peggiore.

Una strategia randomizzata è utile tipicamente quando vi sono molti modi in cui un algoritmo può procedere, ma è difficile determinare quello che promette di essere migliore. Se molte alternative sono buone, sceglierne semplicemente una in modo casuale può portare a una buona strategia. Spesso un algoritmo deve fare molte scelte durante la sua esecuzione. Se i benefici delle scelte buone hanno maggior peso dei costi delle scelte cattive, una selezione casuale tra scelte buone e cattive può portare a un algoritmo efficiente. Abbiamo notato nel

paragrafo 8.2 che un misto di partizioni buone e cattive produce un buon tempo di esecuzione per il quicksort, e quindi si capisce che versioni randomizzate dell'algoritmo dovrebbero comportarsi bene.

Modificando la procedura PARTITION, si può progettare un'altra versione randomizzata del quicksort che usi la seguente strategia di scelta casuale. Ad ogni passo dell'algoritmo quicksort, prima di partizionare l'array, si scambia l'elemento $A[p]$ con un elemento scelto a caso in $A[p \,.\,.\, r]$. Questa modifica assicura che l'elemento perno $x = A[p]$ sia in modo equiprobabile uno qualunque degli $r - p + 1$ elementi del sottoarray. Di conseguenza ci si aspetta che la suddivisione dell'array in input sia, in media, ragionevolmente ben bilanciata. Anche l'algoritmo randomizzato basato sulla permutazione casuale dell'array di input funziona bene in media, ma è un po' più difficile da analizzare di questa versione.

Le modifiche da apportare a PARTITION e QUICKSORT sono poche. Nella nuova procedura di partizionamento, basta effettuare lo scambio prima dell'effettivo partizionamento:

```
RANDOMIZED-PARTITION (A, p, r)
    i ← RANDOM (p, r)
    scambia A[p] ↔ A[i]
    return PARTITION (A, p, r)
```

Il nuovo quicksort chiama adesso RANDOMIZED-PARTITION invece di PARTITION:

```
RANDOMIZED-QUICKSORT (A, p, r)
    if p < r
      then q ← RANDOMIZED-PARTITION (A, p, r)
           RANDOMIZED-QUICKSORT (A, p, r)
           RANDOMIZED-QUICKSORT (A, q + 1, r)
```

Si analizzerà l'algoritmo nel prossimo paragrafo.

Esercizi

8.3-1 Perché analizzare le prestazioni di un algoritmo randomizzato nel caso medio e non nel caso peggiore?

8.3-2 Durante l'esecuzione della procedura RANDOMIZED-QUICKSORT, quante chiamate vengono fatte al generatore di numeri casuali RANDOM nel caso peggiore? Come cambia la risposta nel caso migliore?

* *8.3-3* Descrivere una realizzazione della procedura RANDOM (a, b) che usi solo lanci di una moneta equa. Qual è il tempo di esecuzione atteso della procedura?

* *8.3-4* Scrivere una procedura randomizzata con tempo di esecuzione $\Theta(n)$ che prenda come input un array $A[1 \,.\,.\, n]$ ed esegua una permutazione casuale sugli elementi dell'array.

8.4 Analisi del quicksort

Il paragrafo 8.2 fornisce spiegazioni intuitive sul comportamento del quicksort nel caso peggiore e sul perché ci si aspetta che sia eseguito velocemente. In questo paragrafo, si analizzerà il suo comportamento in modo più rigoroso. Si comincerà con l'analisi del caso peggiore che vale tanto per QUICKSORT che per RANDOMIZED-QUICKSORT, e si concluderà con un'analisi del caso medio di RANDOMIZED-QUICKSORT.

8.4.1 Analisi del caso peggiore

Nel paragrafo 8.2 si è visto che il partizionamento peggiore ad ogni livello di ricorrenza di quicksort produce un tempo di esecuzione $\Theta(n^2)$, che, intuitivamente, è il caso peggiore per il tempo di esecuzione dell'algoritmo. Questa affermazione sarà ora dimostrata.

Usando il metodo di sostituzione (si veda il paragrafo 4.1), si può mostrare che il tempo di esecuzione di quicksort è $O(n^2)$. Sia $T(n)$ il tempo nel caso peggiore della procedura QUICKSORT su un input di dimensione n. Si ha la ricorrenza

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n) , \tag{8.1}$$

dove il parametro q varia tra 1 e $n-1$ perché la procedura PARTITION produce due regioni, ognuna con dimensione almeno 1. Si può tentare la soluzione $T(n) \le cn^2$ per qualche costante c. Sostituendo nella (8.1), si ottiene

$$\begin{aligned} T(n) &\le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) \\ &= c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n) . \end{aligned}$$

L'espressione $q^2 + (n-q)^2$, nell'intervallo $1 \le q \le n-1$, raggiunge un valore massimo in uno dei due estremi, come si può verificare dal fatto che la derivata seconda dell'espressione rispetto a q è positiva (si veda l'Esercizio 8.4-2).
Da ciò si ha il limite $\max_{1 \le q \le n-1}(q^2 + (n-q)^2) \le 1^2 + (n-1)^2 = n^2 - 2(n-1)$. Proseguendo con il confronto su $T(n)$ si ottiene

$$\begin{aligned} T(n) &\le cn^2 - 2c(n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

perché si può scegliere la costante c sufficientemente grande in modo che il termine $2c(n-1)$ superi il termine $\Theta(n)$. Pertanto il tempo di esecuzione di quicksort nel caso peggiore è $\Theta(n^2)$.

8.4.2 Analisi del caso medio

Si è già data una spiegazione intuitiva sul perché il tempo di esecuzione di RANDOMIZED-QUICKSORT nel caso medio sia $\Theta(n \lg n)$: se la suddivisione indotta da RANDOMIZED-PARTITION pone una frazione costante di elementi su un lato della partizione, allora l'albero di ricorrenza ha profondità $\Theta(\lg n)$ e viene eseguito un lavoro $\Theta(n)$ per ognuno di questi $\Theta(\lg n)$ livelli. Si può analizzare il tempo di esecuzione atteso di RANDOMIZED-QUICKSORT comprendendo il comportamento della procedura di partizionamento. Si può quindi sviluppare una ricorrenza

per il tempo medio richiesto per ordinare un array di n elementi e risolvere questa ricorrenza per limitare il tempo di esecuzione atteso. Come parte del processo di risoluzione della ricorrenza, si deriveranno limiti stretti per un'interessante sommatoria.

Analisi del partizionamento

Cominciamo con qualche osservazione sull'operazione PARTITION. Quando PARTITION è chiamata nella linea 3 della procedura RANDOMIZED-QUICKSORT, l'elemento $A[p]$ è già stato scambiato con un elemento scelto a caso in $A[p \,..\, r]$. Per semplificare l'analisi, si assuma che tutti i numeri in input siano diversi. Se non tutti i numeri sono diversi, è ancora vero che il tempo di esecuzione di quicksort nel caso medio è $O(n \lg n)$, ma dimostrarlo richiede un'analisi più complicata di quella presentata qui.

La prima osservazione è che il valore di q restituito da PARTITION dipende solo dal rango di $x = A[p]$ rispetto agli elementi in $A[p \,..\, r]$. (Il ***rango*** di un numero in un insieme è il numero di elementi minori o uguali ad esso.) Sia $n = r - p + 1$ il numero di elementi in $A[p \,..\, r]$: scambiando $A[p]$ con un elemento scelto a caso in $A[p \,..\, r]$ vi è una probabilità di $1/n$ che $\text{rango}(x) = i$ per $i = 1, 2, \ldots, n$.

Calcoliamo ora le probabilità dei vari risultati del partizionamento. Se $\text{rango}(x) = 1$, allora la prima volta che si esegue il ciclo while nelle linee 4-11 di PARTITION, l'indice i si ferma a $i = p$ e l'indice j si ferma a $j = p$. Di conseguenza, quando è restituito $q = j$, il lato "basso" della partizione contiene il solo elemento $A[p]$. Ciò accade con probabilità $1/n$ dato che questa è la probabilità che $\text{rango}(x) = 1$.

Se $\text{rango}(x) \ge 2$, allora vi è almeno un elemento più piccolo di $x = A[p]$. Di conseguenza, la prima volta che si esegue il ciclo while, l'indice i si ferma a $i = p$ ma l'indice j si ferma prima di raggiungere p. Viene quindi fatto uno scambio con $A[p]$ per metterlo nel lato alto della partizione. Quando PARTITION termina, ognuno dei $\text{rango}(x) - 1$ elementi nel lato basso della partizione è strettamente minore di x. Pertanto, per ciascun $i = 1, 2, \ldots, n-1$, se $\text{rango}(x) \ge 2$, la probabilità che il lato basso della partizione abbia i elementi è $1/n$.

Dalla combinazione dei due casi si può concludere che la dimensione $q = p + 1$ del lato basso della partizione è 1 con probabilità $2/n$ e che la dimensione è i con probabilità $1/n$ per $i = 2, 3, \ldots, n-1$.

Una ricorrenza per il caso medio

Si stabilisce ora una ricorrenza per il tempo di esecuzione atteso di RANDOMIZED-QUICKSORT. Sia $T(n)$ il tempo medio richiesto per ordinare un array di input di n elementi. Una chiamata a RANDOMIZED-QUICKSORT su un array di un solo elemento richiede tempo costante e quindi si ha $T(1) = \Theta(1)$. Una chiamata a RANDOMIZED-QUICKSORT su un array $A[1 \,..\, n]$ di lunghezza n richiede tempo $\Theta(n)$ per partizionare l'array. La procedura PARTITION restituisce un indice q, e RANDOMIZED-QUICKSORT è chiamata ricorsivamente sui sottoarray di lunghezza q e $n - q$. Di conseguenza il tempo medio per ordinare un array di lunghezza n può essere espresso come

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1} (T(q) + T(n-q))\right) + \Theta(n) . \tag{8.2}$$

Il valore di q ha una distribuzione quasi uniforme, tranne per il valore $q = 1$ che è probabile il doppio degli altri, come notato prima.

Usando il fatto che $T(1) = \Theta(1)$ e $T(n-1) = O(n^2)$ dall'analisi del caso peggiore, si ha

$$\begin{aligned}\frac{1}{n}(T(1) + T(n-1)) &= \frac{1}{n}(\Theta(1) + O(n^2)) \\ &= O(n) ,\end{aligned}$$

e il termine $\Theta(n)$ nell'equazione (8.2) può perciò assorbire l'espressione $\frac{1}{n}(T(1) + T(n-1))$. Si può quindi riscrivere la ricorrenza (8.2) come

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n) . \tag{8.3}$$

Si osservi che per $k = 1, 2, \ldots, n-1$, ogni termine $T(k)$ della somma compare una volta come $T(q)$ e una volta come $T(n-q)$. Fondendo i due termini della somma si ottiene

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) . \tag{8.4}$$

Soluzione della ricorrenza

Si può risolvere la ricorrenza (8.4) usando il metodo di sostituzione. Si assuma induttivamente che $T(n) \le an \lg n + b$ per certe costanti $a > 0$ e $b > 0$ ancora da stabilire. Si possono scegliere a e b sufficientemente grandi in modo che $an \lg n + b$ sia più grande di $T(1)$. Quindi per $n > 1$, si ha la sostituzione

$$\begin{aligned}T(n) &= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \\ &\le \frac{2}{n}\sum_{k=1}^{n-1}(ak \lg k + b) + \Theta(n) \\ &= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n) .\end{aligned}$$

Si mostrerà tra un attimo che la sommatoria nell'ultima riga può essere limitata da

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 . \tag{8.5}$$

Usando questa limitazione, si ottiene

$$\begin{aligned}T(n) &\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \\ &\le an \lg n - \frac{a}{4}n + 2b + \Theta(n) \\ &= an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \\ &\le an \lg n + b ,\end{aligned}$$

poiché si può scegliere a sufficientemente grande così che $\frac{a}{4}n$ superi $\Theta(n) + b$. Si conclude che il tempo medio di esecuzione del quicksort è $O(n \lg n)$.

Limiti stretti sulla sommatoria

Rimane da provare il limite (8.5) sulla sommatoria

$$\sum_{k=1}^{n-1} k \lg k .$$

Poiché ogni termine è al più $n \lg n$, si ha la disuguaglianza

$$\sum_{k=1}^{n-1} k \lg k \le n^2 \lg n ,$$

che definisce un limite stretto a meno di un fattore costante. Questo limite però non è sufficientemente forte da risolvere la ricorrenza con $T(n) = O(n \lg n)$. In particolare, si ha bisogno di un limite di $\frac{1}{2}n^2 \lg n - \Omega(n^2)$ perché la soluzione della ricorrenza possa essere ottenuta.

Si può determinare questo limite sulla sommatoria dividendola in due parti, come discusso nel paragrafo 3.2, ottenendo

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k .$$

Il $\lg k$ nella prima sommatoria della parte destra è superiormente limitato da $\lg(n/2) = \lg n - 1$; il $\lg k$ nella seconda sommatoria è superiormente limitato da $\lg n$. Quindi si ha

$$\begin{aligned}\sum_{k=1}^{n-1} k \lg k &\le (\lg n - 1)\sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\ &= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\ &\le \frac{1}{2}n(n-1)\lg n - \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2} \\ &\le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\end{aligned}$$

se $n \ge 2$. Con ciò è dimostrata la disuguaglianza (8.5).

Esercizi

8.4-1 Mostrare che il tempo di esecuzione del quicksort nel caso migliore è $\Omega(n \lg n)$.

8.4-2 Mostrare che $q^2 + (n-q)^2$ raggiunge un valore massimo per $q = 1, 2, \ldots, n-1$ quando $q = 1$ o $q = n-1$.

8.4-3 Mostrare che il tempo di esecuzione atteso per il RANDOMIZED-QUICKSORT è $\Omega(n \lg n)$.

8.4-4 Il tempo di esecuzione del quicksort può essere migliorato in pratica traendo vantaggio dal fatto che l'algoritmo insertion sort ha il tempo di esecuzione più veloce quando il suo input è "quasi" ordinato. La procedura è la seguente: il quicksort su un sottoarray con meno di k elementi lo restituisce senza ordinarlo: quando la chiamata principale al quicksort è terminata, si esegue l'insertion sort su tutto l'array per completare il processo di ordinamento. Mostrare che questo algoritmo di ordinamento ha un tempo di esecuzione atteso $O(nk + n \lg(n/k))$. Come, in teoria e in pratica, dovrebbe essere scelto k?

* *8.4-5* Provare l'identità

$$\int x \ln x \, dx = \frac{1}{2}x^2 \ln x - \frac{1}{4}x^2 ,$$

e quindi usare il metodo di approssimazione con integrali per fornire una maggiorazione della sommatoria $\sum_{k=1}^{n-1} k \lg k$ che sia più stretta della (8.5).

* *8.4-6* Prendere in considerazione una modifica alla procedura PARTITION che scelga in modo casuale tre elementi nell'array A ed esegua il partizionamento rispetto al mediano fra i tre valori. Approssimare con una funzione di α la probabilità di ottenere, nel caso peggiore, una suddivisione di α ad $(1 - \alpha)$ con $0 < \alpha < 1$.

Problemi

8-1 Correttezza della procedura PARTITION

Dimostrare formalmente che la procedura PARTITION del paragrafo 8.1 è corretta. Dimostrare i seguenti punti:

a. Gli indici i e j non si riferiscono mai a un elemento di A fuori dall'intervallo $[p \,.\,.\, r]$.

b. L'indice j è diverso da r quando PARTITION termina (cioè la suddivisione non è mai banale).

c. Ogni elemento di $A[p \,.\,.\, j]$ è minore o uguale ad ogni elemento di $A[j + 1 \,.\,.\, r]$ quando PARTITION termina.

8-2 Algoritmo di partizionamento di Lomuto

Considerare la seguente variazione di PARTITION, dovuta a N. Lomuto. Per partizionare $A[p \,.\,.\, r]$, questa versione costruisce due regioni, $A[p \,.\,.\, i]$ e $A[i + 1 \,.\,.\, j]$, tali che ogni elemento nella prima regione sia minore o uguale a $x = A[r]$ e ogni elemento nella seconda regione sia più grande di x.

```
LOMUTO-PARTITION (A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r
```



```
    do if A[j] ≤ x
          then i ← i + 1
               scambia A[i] ↔ A[j]
if i < r
    then return i
    else return i − 1
```

a. Mostrare che la procedura LOMUTO-PARTITION è corretta.

b. Qual è il numero massimo di volte che un elemento può essere spostato da PARTITION e da LOMUTO-PARTITION?

c. Mostrare che la procedura LOMUTO-PARTITION, come la PARTITION, viene eseguita in tempo $\Theta(n)$ su un sottoarray di n elementi.

d. Come è influenzato il tempo di esecuzione di QUICKSORT, quando tutti i valori di input sono uguali, se si sostituisce PARTITION con LOMUTO-PARTITION?

e. Definire una procedura RANDOMIZED-LOMUTO-PARTITION che scambia $A[r]$ con un elemento scelto a caso in $A[p \,.\,.\, r]$ e quindi chiama la LOMUTO-PARTITION. Mostrare che la probabilità che un dato valore q sia restituito dalla RANDOMIZED-LOMUTO-PARTITION è uguale alla probabilità che il valore $p + r - q$ sia restituito dalla RANDOMIZED-PARTITION.

8-3 Algoritmo Stooge Sort

I professori Gatto e Volpe hanno proposto il seguente "elegante" algoritmo di ordinamento:

```
STOOGE-SORT (A, i, j)
if A[i] > A[j]
    then scambia A[i] ↔ A[j]
if i + 1 ≥ j
    then return
k ← ⌊(j − i + 1)/3⌋          ▷ Suddivisione in tre parti.
STOOGE-SORT (A, i, j − k)    ▷ Si ordinano i primi due terzi dell'array.
STOOGE-SORT (A, i + k, j)    ▷ Si ordinano gli ultimi due terzi dell'array.
STOOGE-SORT (A, i, j − k)    ▷ Si ordinano di nuovo i primi due terzi dell'array.
```

a. Mostrare che STOOGE-SORT $(A, 1, length[A])$ ordina correttamente l'array $A[1 \,.\,.\, n]$ di input, dove $n = length[A]$.

b. Fornire la ricorrenza per il tempo di esecuzione di STOOGE-SORT nel caso peggiore ed un limite asintotico stretto (notazione Θ) sempre per il tempo di esecuzione del caso peggiore.

c. Confrontare il tempo di esecuzione del caso peggiore di STOOGE-SORT con quello dell'insertion sort, merge sort, heapsort e quicksort. I professori hanno ancora diritto ad essere chiamati tali?

8-4 *Profondità della pila per il quicksort*

L'algoritmo QUICKSORT nel paragrafo 8.1 contiene due chiamate ricorsive a se stesso. Dopo la chiamata a PARTITION, sono ordinati ricorsivamente prima il sottoarray di sinistra, e poi quello di destra. La seconda chiamata ricorsiva nella procedura QUICKSORT in realtà non è necessaria; può essere evitata usando una struttura di controllo iterativa. Questa tecnica è fornita da buoni compilatori in modo automatico. Si consideri la seguente versione di quicksort, che simula questa tecnica (chiamata in inglese ***tail recursion***).

```
QUICKSORT' (A, p, r)
    while p < r
        do    ▷ Partiziona e ordina il sottoarray di sinistra
              q ← PARTITION (A, p, r)
              QUICKSORT' (A, p, q)
              p ← q + 1
```

a. Spiegare perché QUICKSORT' (A, 1, *length*[A]) ordina correttamente l'array A.

I compilatori di solito eseguono procedure ricorsive usando una ***pila*** che contiene le informazioni relative ad ogni chiamata ricorsiva, inclusi i valori dei parametri. Le informazioni sulla chiamata più recente sono sulla cima della pila mentre quelle della chiamata iniziale sono in fondo. Quando una procedura è chiamata, le sue informazioni vengono ***inserite*** sulla pila; quando questa termina le sue informazioni vengono ***estratte***. Poiché si assume che di fatto i parametri relativi all'array siano rappresentati da puntatori, le informazioni per ogni chiamata di procedura richiedono spazio $O(1)$ nella pila. La ***profondità della pila*** è la massima quantità di spazio usato nella pila in qualunque momento dell'esecuzione.

b. Descrivere una situazione in cui la profondità della pila di QUICKSORT' è $\Theta(n)$ su un array di input di n elementi.

c. Modificare il codice di QUICKSORT' in modo che la profondità della pila nel caso peggiore sia $\Theta(\lg n)$.

8-5 *Partizione sul mediano-fra-tre*

Un modo per migliorare la procedura RANDOMIZED-QUICKSORT è di usare come perno per la partizione un elemento x scelto più attentamente che prendendone uno a caso dal sottoarray. Un approccio comune è il metodo del ***mediano-fra-tre***: si sceglie come x l'elemento mediano su un insieme di tre elementi selezionati in modo casuale nel sottoarray. Per questo problema si assume che gli elementi dell'array di input $A[1 \ldots n]$ siano distinti ed $n \geq 3$. Sia $A'[1 \ldots n]$ l'array di output ordinato. Usando il metodo mediano-fra-tre per scegliere l'elemento perno x, definire $p_i = \Pr\{x = A'[i]\}$.

a. Fornire una formula esatta per descrivere p_i come funzione di n ed i, per $i = 2, 3, \ldots, n-1$. (Si noti che $p_1 = p_n = 0$).

b. Di quanto è aumentata la probabilità di scegliere $x = A'[\lfloor (n+1)/2 \rfloor]$, cioè il mediano in $A[1 \ldots n]$, rispetto alla realizzazione usuale dell'algoritmo? Si assuma che $n \to \infty$, e si fornisca la valutazione al limite di queste probabilità.

c. Se si definisce come "buona" suddivisione lo scegliere $x = A'[i]$, con $n/3 \leq i \leq 2n/3$, di quanto cresce la probabilità di avere una buona suddivisione rispetto alla realizzazione usuale dell'algoritmo? (*Suggerimento*: si approssimi la somma con un integrale).

d. Dedurre che il metodo mediano-fra-tre influenza solo il fattore costante del tempo $\Omega(n \lg n)$ di quicksort.

Note al Capitolo

La procedura quicksort fu proposta da Hoare [98]. Sedgewick [174] fornisce una buona trattazione dei dettagli di realizzazione dell'algoritmo e della loro importanza. I vantaggi degli algoritmi randomizzati furono evidenziati da Rabin [165].

Ordinamento in tempo lineare

9

Sono stati appena esaminati alcuni algoritmi che ordinano n numeri in tempo $O(n \lg n)$. Il merge sort e lo heapsort raggiungono questo limite superiore nel caso peggiore; il quicksort lo raggiunge in media. Inoltre, per ognuno di questi algoritmi, si può produrre una sequenza di n numeri di input in modo che l'algoritmo venga eseguito in tempo $\Omega(n \lg n)$.

Questi algoritmi condividono un'interessante proprietà: *l'ordinamento che determinano è basato solo sui confronti tra elementi di input*. Questi algoritmi di ordinamento sono chiamati ***ordinamenti per confronti***. Tutti gli algoritmi di ordinamento presentati finora sono ordinamenti per confronti.

Nel paragrafo 9.1 sarà dimostrato che qualunque ordinamento per confronti deve eseguire, nel caso peggiore, $\Omega(n \lg n)$ confronti per ordinare una sequenza di n elementi. Per cui, il merge sort e lo heapsort sono asintoticamente ottimi, e non esiste alcun ordinamento per confronti che sia più veloce, se non per un fattore costante.

I paragrafi 9.2, 9.3 e 9.4 esaminano tre algoritmi di ordinamento – counting sort, radix sort e bucket sort – che vengono eseguiti in tempo lineare. È superfluo dire che questi algoritmi usano operazioni diverse dal confronto per determinare l'ordinamento; di conseguenza, per essi il limite inferiore $\Omega(n \lg n)$ non vale.

9.1 Limiti inferiori per l'ordinamento

In un ordinamento per confronti, si usa solo il confronto tra elementi per ottenere informazioni sull'ordine della sequenza di input $\langle a_1, a_2, \ldots, a_n \rangle$. Cioè, dati due elementi a_i e a_j, per determinarne l'ordinamento relativo, si esegue uno dei seguenti confronti: $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$. Non si possono esaminare i valori degli elementi o ottenere informazioni sul loro ordine in alcun altro modo.

In questo paragrafo si assume, senza perdita di generalità, che tutti gli elementi di input siano distinti. Data questa ipotesi, i confronti della forma $a_i = a_j$ sono inutili, e così si può assumere che non vengano fatti. Si noti inoltre che i confronti $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ e $a_i < a_j$ sono tutti equivalenti, nel senso che forniscono la stessa informazione sull'ordinamento relativo tra a_i e a_j. Quindi si assume che tutti i confronti abbiano la forma $a_i \leq a_j$.

Figura 9.1 *L'albero di decisione per l'insertion sort che opera su tre elementi. Vi sono 3! = 6 possibili permutazioni degli elementi di input, quindi l'albero di decisione deve avere almeno 6 foglie.*

Il modello ad albero di decisione

Gli ordinamenti per confronti possono essere visti in modo astratto in termini di ***alberi di decisione***. Un albero di decisione rappresenta il confronto eseguito da un algoritmo di ordinamento quando opera su un input di una data dimensione. Il controllo, i movimenti dei dati e tutti gli altri aspetti dell'algoritmo vengono ignorati. La figura 9.1 mostra l'albero di decisione corrispondente all'algoritmo di insertion sort del paragrafo 1.1 che opera su una sequenza di input di tre elementi.

In un albero di decisione, ogni nodo interno è etichettato con $a_i : a_j$ per qualche i e j nell'intervallo $1 \le i, j \le n$, dove n è il numero di elementi nella sequenza di input. Ogni foglia è etichettata da una permutazione $\langle \pi(1), \pi(2), \ldots, \pi(n)\rangle$. (Si vedano nel paragrafo 6.1 i richiami sulle permutazioni). L'esecuzione di un algoritmo di ordinamento corrisponde a tracciare un cammino sull'albero di decisione dalla radice ad una foglia. In ogni nodo interno viene eseguito un confronto $a_i \le a_j$; il sottoalbero sinistro impone i successivi confronti per $a_i \le a_j$, mentre il sottoalbero destro impone i confronti per $a_i > a_j$. Quando si arriva ad una foglia, l'algoritmo ha stabilito l'ordinamento $a_{\pi(1)} \le a_{\pi(2)} \le \ldots \le a_{\pi(n)}$. Perché l'algoritmo funzioni in modo appropriato, ognuna delle $n!$ permutazioni di n elementi deve comparire in una delle foglie dell'albero di decisione.

Un limite inferiore per il caso peggiore

La lunghezza del cammino più lungo di un albero di decisione dalla radice ad una qualunque delle sue foglie rappresenta il numero di confronti che l'algoritmo di ordinamento esegue nel caso peggiore, numero che, di conseguenza, corrisponde all'altezza dell'albero di decisione. Un limite inferiore sull'altezza degli alberi di decisione è allora un limite inferiore sul tempo di esecuzione di qualunque algoritmo di ordinamento per confronti. Il seguente teorema stabilisce tale limite inferiore.

Teorema 9.1

Qualunque albero di decisione che ordina n elementi ha altezza $\Omega(n \lg n)$.

Dimostrazione. *Si consideri un albero di decisione di altezza h che ordina n elementi. Poiché vi sono $n!$ permutazioni di n elementi, dove ogni permutazione rappresenta un ordinamento diverso degli elementi, l'albero deve avere almeno $n!$ foglie.*

Poiché un albero binario di altezza h non ha più di 2^h foglie, si ha

$$n! \le 2^h ,$$

che, passando ai logaritmi, implica

$$h \ge \lg(n!) ,$$

poiché la funzione lg è monotona crescente. Dall'approssimazione di Stirling (2.11), si ha

$$n! > \left(\frac{n}{e}\right)^n ,$$

dove $e = 2.71828\ldots$ è la base dei logaritmi naturali; da cui

$$\begin{aligned} h &\ge \lg\left(\frac{n}{e}\right)^n \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n) . \end{aligned}$$

■

Corollario 9.2

Il merge sort e lo heapsort sono ordinamenti per confronti asintoticamente ottimi.

Dimostrazione. I limiti superiori $O(n \lg n)$ sui tempi di esecuzione degli algoritmi heapsort e merge sort corrispondono al limite inferiore $\Omega(n \lg n)$ nel caso peggiore descritto nel Teorema 9.1. ■

Esercizi

9.1-1 Qual è la profondità minima possibile di una foglia nell'albero di decisione di un algoritmo di ordinamento?

9.1-2 Ottenere limiti asintoticamente stretti su $\lg(n!)$ senza usare l'approssimazione di Stirling. Calcolare la sommatoria $\sum_{k=1}^{n} \lg k$ usando invece le tecniche presentate nel paragrafo 3.2.

9.1-3 Mostrare che non vi sono ordinamenti per confronti il cui tempo di esecuzione sia lineare su almeno metà degli $n!$ input di lunghezza n. Cosa si può dire di una frazione di $1/n$ degli input di lunghezza n? E cosa si può dire di una frazione di $1/2^n$?

9.1-4 Il professor Salomone sostiene che il limite inferiore $\Omega(n \lg n)$ per ordinare n numeri non è applicabile all'ambiente del suo calcolatore, in cui il flusso di controllo di un programma, dopo ogni singolo confronto $a_i : a_j$, si può dividere in tre modi, a seconda che $a_i < a_j$, $a_i = a_j$, $a_i > a_j$. Mostrare che il professore è in errore provando che il numero di confronti a tre vie richiesto per ordinare n elementi è ancora $\Omega(n \lg n)$.

9.1-5 Provare che sono necessari $2n - 1$ confronti per eseguire, nel caso peggiore, la fusione (o merge) di due liste ordinate contenenti ognuna n elementi.

9.1-6 Sia data una sequenza di n elementi da ordinare, costituita di n/k sottosequenze, ognuna contenente k elementi. Gli elementi in una data sottosequenza sono tutti più piccoli degli elementi della sottosequenza successiva e più grandi degli elementi della precedente. Così, per ordinare l'intera sequenza di lunghezza n è necessario soltanto ordinare i k elementi di ognuna delle n/k sottosequenze. Mostrare che $\Omega(n \lg k)$ è un limite inferiore sul numero di confronti necessari a risolvere questa variante del problema di ordinamento. (*Suggerimento*: non è rigoroso combinare semplicemente i limiti inferiori delle singole sottosequenze.)

9.2 Counting sort

Il ***counting sort*** si basa sull'ipotesi che ognuno degli n elementi di input sia un intero nell'intervallo da 1 a k, per qualche intero k. Quando $k = O(n)$, l'ordinamento viene eseguito in tempo $O(n)$.

L'idea di base del counting sort è di determinare, per ogni elemento x in input, il numero di elementi minori di x. Questa informazione può essere usata per porre l'elemento x direttamente nella sua posizione nell'array di output. Per esempio, se vi sono 17 elementi minori di x, allora x va messo nella posizione 18 dell'output. Questo schema deve essere modificato leggermente per gestire la situazione in cui alcuni elementi abbiano lo stesso valore, infatti non si vuole metterli tutti nella stessa posizione.

Nel codice del counting sort, si assume che l'input sia un array $A[1 \ldots n]$ e che $length[A] = n$. Sono richiesti altri due array: l'array $B[1 \ldots n]$ mantiene l'output ordinato e l'array $C[1 \ldots k]$ fornisce la memoria di lavoro temporanea.

```
COUNTING-SORT(A, B, k)
1   for i ← 1 to k
2       do C[i] ← 0
3   for j ← 1 to length[A]
4       do C[A[j]] ← C[A[j]] + 1
5   ▷ C[i] contiene ora il numero di elementi uguali a i.
6   for i ← 2 to k
7       do C[i] ← C[i] + C[i − 1]
8   ▷ C[i] contiene ora il numero di elementi minori o uguali a i.
9   for j ← length[A] downto 1
10      do B[C[A[j]]] ← A[j]
11         C[A[j]] ← C[A[j]] − 1
```

Il counting sort è illustrato nella figura 9.2. Dopo l'inizializzazione nelle linee 1-2, si analizza ogni elemento di input nelle linee 3-4. Se il valore di un elemento di input è i, si incrementa $C[i]$. Così, dopo le linee 3-4, $C[i]$ contiene il numero di elementi di input uguali

Figura 9.2 *L'esecuzione di* COUNTING-SORT *su un array di input $A[1, \ldots, 8]$, dove ogni elemento di A è un intero positivo non più grande di $k = 6$.* (a) *L'array A e l'array ausiliario C dopo la linea 4.* (b) *L'array C dopo la linea 7.* (c)-(e) *L'array di output B e l'array ausiliario C rispettivamente dopo una, due e tre iterazioni del ciclo di linee 9-11. Solo gli elementi grigio chiaro dell'array B sono stati riempiti.* (f) *L'array finale di output B ordinato.*

ad i per ciascun intero $i = 1, 2, \ldots, k$. Nelle linee 6-7, si determina per ciascun $i = 1, 2, \ldots, k$, quanti elementi dell'input sono minori o uguali ad i; ciò viene fatto determinando la somma cumulata degli elementi dell'array C.

Infine, nelle linee 9-11, si pone ogni elemento $A[j]$ nell'array di output B nella sua corretta posizione ordinata. Se tutti gli n elementi sono distinti, allora la prima volta che viene eseguita la linea 9, per ciascun $A[j]$, il valore $C[A[j]]$ è la posizione finale corretta di $A[j]$ nell'array di output: infatti vi sono $C[A[j]]$ elementi minori o uguali ad $A[j]$. Poiché gli elementi potrebbero non essere distinti, si decrementa $C[A[j]]$ ogni volta che si pone un valore $A[j]$ nell'array B; ciò fa sì che il prossimo elemento di input con un valore uguale ad $A[j]$, se esiste, vada nella posizione immediatamente precedente $A[j]$ nell'array di output.

Quanto tempo richiede il counting sort? Il ciclo **for** nelle linee 1-2 impiega un tempo $O(k)$, il ciclo **for** nelle linee 3-4 impiega un tempo $O(n)$, il ciclo **for** nelle linee 6-7 impiega un tempo $O(k)$ e il ciclo **for** nelle linee 9-11 impiega un tempo $O(n)$, da cui il tempo totale è $O(k + n)$. In pratica, si usa di solito il counting sort quando si ha $k = O(n)$, nel qual caso il tempo di esecuzione è $O(n)$.

Il counting sort abbatte il limite inferiore $\Omega(n \lg n)$ dimostrato nel paragrafo 9.1 perché non è un ordinamento per confronti. In effetti, nel codice non viene mai eseguito alcun confronto tra gli elementi di input. Il counting sort usa, invece, i valori effettivi degli elementi per indicare direttamente un elemento all'interno di un array. Il limite inferiore $\Omega(n \lg n)$ per gli ordinamenti non è valido quando si abbandona il modello degli ordinamenti per confronti.

Una proprietà importante del counting sort è che è ***stabile***: elementi con lo stesso valore compaiono nell'array di output nello stesso ordine in cui compaiono in quello di input. Cioè lo spareggio tra due elementi con lo stesso valore avviene secondo la regola per cui qualunque numero compaia per primo nell'array di input appare per primo nell'array di output. Naturalmente la proprietà di stabilità è importante solo quando i dati satellite sono trasferiti insieme con gli elementi da ordinare. Si vedrà l'importanza della proprietà di stabilità nel prossimo paragrafo.

Esercizi

9.2-1 Usando la figura 9.2 come modello, mostrare l'esecuzione di COUNTING-SORT sull'array $A = \langle 7, 1, 3, 1, 2, 4, 5, 7, 2, 4, 3 \rangle$.

9.2-2 Dimostrare che COUNTING-SORT è stabile.

9.2-3 Si supponga che la linea 9 del ciclo **for** della procedura COUNTING-SORT sia riscritta come

9 **for** $j \leftarrow 1$ **to** *length*[A]

Mostrare che l'algoritmo funziona ancora in modo appropriato. L'algoritmo modificato è stabile?

9.2-4 Si supponga che l'output dell'algoritmo di ordinamento sia un flusso di dati su uno schermo grafico. Modificare COUNTING-SORT per produrre l'output ordinato senza usare sostanzialmente memoria addizionale oltre ad A e C. (*Suggerimento*: collegare gli elementi di A con la stessa chiave in liste concatenate. Dov'è un posto "libero" per tenere i puntatori agli elementi della lista?)

9.2-5 Descrivere un algoritmo che, dati n interi nell'intervallo da 1 a k, esegua un'analisi preliminare del suo input e quindi risponda in tempo $O(1)$ a qualunque domanda su quanti degli n interi cadano nell'intervallo $[a \ldots b]$. L'algoritmo dovrebbe usare per l'analisi preliminare un tempo $O(n + k)$.

9.3 Radix sort

Il *radix sort* è l'algoritmo usato dalle macchine "ordinatrici di schede" che si trovano ora solo nei musei dei calcolatori. Le schede sono suddivise in 80 colonne e ogni colonna può essere perforata in una delle 12 righe. La macchina ordinatrice può essere "programmata" meccanicamente perché esamini una data colonna di ogni scheda e inserisca la scheda in uno di 12 contenitori a seconda di quale riga è stata perforata. Un operatore può quindi raccogliere le schede contenitore per contenitore, in modo che le schede con la prima riga perforata siano messe sopra le schede con la seconda riga perforata e così via.

Per le cifre decimali, in ogni colonna sono usate solo 10 righe. (Le altre due sono usate per codificare caratteri non numerici.) Un numero di d cifre dovrebbe allora occupare un campo di d colonne. Poiché l'ordinatrice di schede può controllare solo una colonna alla volta, il problema di ordinare n schede contenenti ciascuna un numero di d cifre richiede un algoritmo di ordinamento.

Intuitivamente, si potrebbe desiderare di ordinare i numeri rispetto alla loro cifra *più significativa*, ordinare poi ricorsivamente ognuno dei contenitori risultanti e quindi combinare i blocchi in modo ordinato. Purtroppo, poiché le schede in 9 dei 10 contenitori devono essere messe da parte per ordinare ognuno dei contenitori, questa procedura genera molti pacchi intermedi di schede dei quali bisogna tenere traccia. (Si veda l'Esercizio 9.3-5).

329		720		720		329
457		355		329		355
657		436		436		436
839	⇒	457	⇒	839	⇒	457
436		657		355		657
720		329		457		720
355		839		657		839
		↑		↑		↑

Figura 9.3 *L'esecuzione di radix sort su una lista di sette numeri di 3 cifre. La prima colonna è l'input. Le altre colonne mostrano la lista dopo un ordinamento su cifre via via più significative. Le frecce verticali indicano la posizione della cifra su cui è stato fatto l'ordinamento per produrre la lista dalla precedente.*

Il radix sort risolve il problema di ordinare le schede in modo controintuitivo ordinando prima sulla cifra *meno significativa*. Le schede sono quindi combinate in un unico blocco, con le schede nel contenitore 0 prima delle schede nel contenitore 1 prima di quelle nel contenitore 2 e così via. Quindi l'intero blocco è ordinato di nuovo sulla seconda cifra meno significativa e ricombinato allo stesso modo. Il processo continua finché le schede sono state ordinate per tutte le d cifre: a quel punto, le schede sono completamente ordinate. Quindi per l'ordinamento sono richieste solo d passate del blocco. La figura 9.3 mostra come opera il radix sort su un blocco di sette numeri di 3 cifre.

La cosa essenziale in quest'algoritmo è che l'ordine delle cifre rimanga stabile. L'ordinamento eseguito da un ordinatore di schede è stabile, ma l'operatore deve stare attento a non cambiare l'ordine delle schede ottenuto, anche se tutte le schede in un contenitore hanno la stessa cifra nella colonna appena esaminata.

In un comune calcolatore, che è una macchina sequenziale ad accesso diretto, il radix sort è talvolta usato per ordinare record di informazioni con chiavi multiple, che comprendono più campi. Per esempio, si potrebbe desiderare di ordinare le date su tre chiavi: anno, mese e giorno. Si potrebbe far eseguire un algoritmo di ordinamento con una funzione di confronto che operi su due date in questo modo: confronta gli anni e, in caso di parità, confronta i mesi e se c'è ancora parità confronta i giorni. Alternativamente, si potrebbero ordinare le informazioni tre volte con un ordinamento stabile: prima sul giorno, poi sul mese e infine sull'anno.

Il codice del radix sort è semplice. La seguente procedura assume che ogni elemento nell'array A di n elementi abbia d cifre, dove la cifra 1 è quella di ordine più basso mentre la cifra d è quella di ordine più alto.

```
RADIX-SORT(A, d)
1  for i ← 1 to d
2      do usa un ordinamento stabile per ordinare l'array A sulla cifra i
```

La correttezza del radix sort viene dimostrata per induzione sulle colonne che devono essere ordinate (si veda l'Esercizio 9.3-3). L'analisi del tempo di esecuzione dipende da quale ordinamento stabile viene usato come algoritmo di ordinamento intermedio. Se ogni cifra è nell'intervallo da 1 a k, e k non è troppo grande, allora viene scelto ovviamente il counting sort. Ogni passo sugli n numeri di d cifre impiega un tempo $\Theta(n + k)$; vi sono d passi, quindi il tempo totale per il radix sort è $\Theta(dn + kd)$. Se d è costante e $k = O(n)$, il radix sort viene eseguito in tempo lineare.

Ad alcuni informatici piace pensare che il numero di bit in una parola di memoria del calcolatore sia $\Theta(\lg n)$. Per praticità, sia $d \lg n$ il numero di bit, dove d è una costante positiva.

Allora se ogni numero da ordinare è rappresentabile in una parola di memoria, lo si può trattare come un numero di d cifre in rappresentazione in base n. Come esempio concreto, si consideri l'ordinamento di 1 milione di numeri di 64 bit. Trattando questi numeri come numeri di quattro cifre rappresentati in base 2^{16}, li si può ordinare in soli 4 passi usando il radix sort. Questa procedura regge in modo favorevole il paragone con un tipico ordinamento per confronti con tempo $\Theta(n \lg n)$, che richiede approssimativamente $\lg n = 20$ operazioni per ciascun numero da ordinare. Purtroppo, la versione del radix sort che usa il counting sort come ordinamento stabile intermedio non ordina in loco, come fanno molti degli ordinamenti per confronti con tempo $\Theta(n \lg n)$. Quindi, se la memoria principale è un aspetto da tenere in forte considerazione, può essere preferibile un algoritmo come il quicksort.

Esercizi

9.3-1 Usando come modello la figura 9.3, illustrare l'operazione di RADIX-SORT sulla seguente lista di parole inglesi: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

9.3-2 Quali dei seguenti algoritmi di ordinamento sono stabili: insertion sort, merge sort, heapsort e quicksort? Descrivere un semplice schema che renda qualunque algoritmo di ordinamento stabile. Quanto tempo e spazio in più richiede lo schema?

9.3-3 Usare l'induzione per provare che il radix sort funziona. In quale punto la dimostrazione richiede l'ipotesi che l'ordinamento intermedio sia stabile?

9.3-4 Mostrare come ordinare n interi presi nell'intervallo da 1 a n^2 in tempo $O(n)$.

* *9.3-5* Nel primo algoritmo di ordinamento delle schede presentato in questo paragrafo, quanti passi sono necessari per ordinare numeri di d cifre decimali nel caso peggiore? Di quanti pacchi di schede dovrebbe tenere traccia un operatore nel caso peggiore?

9.4 Bucket sort

Il *bucket sort* impiega in media un tempo lineare. Come il counting sort, il bucket sort è veloce perché fa delle ipotesi sull'input. Laddove il counting sort assume che l'input consista di interi in un piccolo intervallo, il bucket sort assume che l'input sia generato da un processo casuale che distribuisce gli elementi in modo uniforme sull'intervallo [0,1). (Si veda nel paragrafo 6.2 la definizione di distribuzione uniforme.)

L'idea del bucket sort è di dividere l'intervallo [0,1) in n sottointervalli di uguale dimensione, detti *bucket* (*secchi*), e quindi distribuire gli n numeri nei bucket. Poiché gli elementi dell'input sono uniformemente distribuiti nell'intervallo [0,1), non ci si aspetta che molti numeri cadano nello stesso bucket. Per produrre l'output, si ordinano semplicemente i numeri di ogni bucket e quindi si elencano gli elementi di ogni bucket prendendoli in ordine.

Figura 9.4 *L'esecuzione di* BUCKET-SORT. (a) *L'array in input* $A[1 \ldots 10]$. (b) *L'array* $B[0 \ldots 9]$ *di liste ordinate (bucket) dopo la linea 5 dell'algoritmo. La lista ordinata i contiene i valori nell'intervallo* $[i/10, (i+1)/10)$. *L'output ordinato consiste di una concatenazione ordinata delle liste* $B[0]$, $B[1]$, ..., $B[9]$.

Il codice del bucket sort assume che l'input sia un array di n elementi e che ogni elemento $A[i]$ dell'array soddisfi $0 \le A[i] < 1$. Il codice richiede un array ausiliario $B[0 \ldots n-1]$ di liste concatenate (bucket) e assume che vi sia un meccanismo per mantenere tali liste. (Il paragrafo 11.2 descrive come realizzare le operazioni di base su liste concatenate.)

```
BUCKET-SORT(A)
1  n ← length[A]
2  for i ← 1 to n
3      do inserisci A[i] nella lista B[⌊nA[i]⌋]
4  for i ← 0 to n − 1
5      do ordina la lista B[i] con l'insertion sort
6  concatena insieme le liste B[0], B[1], ..., B[n − 1] in quest'ordine
```

La figura 9.4 mostra l'operazione di bucket sort su un array di input di 10 numeri.

Per mostrare che quest'algoritmo funziona, si considerino due elementi $A[i]$ e $A[j]$. Se questi elementi cadono nello stesso bucket, compariranno nella sequenza di output nell'ordine appropriato, infatti il bucket che li contiene viene ordinato con l'insertion sort. Si supponga ora che cadano in due bucket diversi: $B[i']$ e $B[j']$ rispettivamente, e si assuma senza perdita di generalità che $i' < j'$. Quando le liste di B vengono concatenate nella linea 6, gli elementi del bucket $B[i']$ vengono prima degli elementi di $B[j']$, e così $A[i]$ precederà $A[j]$ nella sequenza di output. Quindi, si deve mostrare che $A[i] \le A[j]$. Assumendo il contrario, si ha

$$\begin{aligned} i' &= \lfloor nA[i] \rfloor \\ &\ge \lfloor nA[j] \rfloor \\ &= j' , \end{aligned}$$

che contraddice l'ipotesi $i' < j'$: quindi il bucket sort è corretto. Per analizzare il tempo di esecuzione, si osservi che tutte le linee eccetto la 5 richiedono un tempo $O(n)$ nel caso

peggiore. Il tempo totale per esaminare tutti i bucket nella linea 5 è $O(n)$, e così l'unica parte interessante dell'analisi è il tempo impiegato dagli insertion sort nella linea 5.

Per analizzare il costo degli insertion sort, sia n_i la variabile casuale che denota il numero di elementi memorizzati nel bucket $B[i]$. Poiché l'insertion sort impiega un tempo quadratico (si veda il paragrafo 1.2), il tempo atteso per ordinare gli elementi nel bucket $B[i]$ è $\mathrm{E}[O(n_i^2)] = O(\mathrm{E}[n_i^2])$. Il tempo totale atteso per ordinare tutti gli elementi in tutti i bucket è allora

$$\sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) = O\left(\sum_{i=0}^{n-1} \mathrm{E}[n_i^2]\right) . \tag{9.1}$$

Per valutare questa sommatoria, si deve determinare la distribuzione di ogni variabile casuale n_i. Vi sono n elementi ed n bucket. La probabilità che un dato elemento cada nel bucket $B[i]$ è $1/n$, poiché ad ogni bucket è assegnato $1/n$ dell'intervallo $[0,1)$. Così, la situazione è analoga a quella dell'esempio del lancio delle palline del paragrafo 6.6.2: vi sono n palline (elementi) ed n contenitori (bucket) e ogni pallina è lanciata indipendentemente con probabilità $p = 1/n$ di cadere in qualunque bucket. Così la probabilità che $n_i = k$ segue la distribuzione binomiale $b(k; n, p)$, che ha valore medio $\mathrm{E}[n_i] = np = 1$ e varianza $\mathrm{Var}[n_i] = np(1-p) = 1 - 1/n$. Per qualunque variabile casuale X, l'equazione (6.30) fornisce

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{Var}[n_i] + \mathrm{E}^2[n_i] \\
&= 1 - \frac{1}{n} + 1^2 \\
&= 2 - \frac{1}{n} \\
&= \Theta(1) .
\end{aligned}$$

Usando questo limite nell'equazione (9.1), si conclude che il tempo atteso per l'ordinamento con l'insertion sort è $O(n)$. Così complessivamente l'algoritmo bucket sort impiega tempo lineare nel caso medio.

Esercizi

9.4-1 Usando come modello la figura 9.4, illustrare l'operazione di BUCKET-SORT sull'array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

9.4-2 Qual'è il tempo di esecuzione dell'algoritmo bucket sort, nel caso peggiore? Quale semplice modifica dell'algoritmo conserva lineare il tempo di esecuzione atteso e impiega tempo $O(n \lg n)$ nel caso peggiore?

* *9.4-3* Siano dati n punti nel cerchio unitario, $p_i = (x_i, y_i)$, tali che $0 < x_i^2 + y_i^2 \le 1$ per $i = 1, 2, \ldots, n$. Si supponga che i punti siano uniformemente distribuiti: cioè, la probabilità di trovare un punto in qualunque regione del cerchio sia proporzionale all'area di quella regione. Progettare un algoritmo, con tempo atteso $\Theta(n)$, per ordinare gli n punti rispetto alla loro $d_i = \sqrt{x_i^2 + y_i^2}$ dall'origine. (*Suggerimento*: definire le dimensioni dei bucket nel BUCKET-SORT in modo da riflettere la distribuzione uniforme dei punti nel cerchio.)

* *9.4-4* Una *funzione di distribuzione di probabilità* $P(x)$ per una variabile casuale X è definita con $P(x) = \Pr\{X \le x\}$. Si supponga che una lista di n numeri abbia una funzione P di distribuzione di probabilità continua calcolabile in tempo $O(1)$. Mostrare come ordinare i numeri con tempo medio lineare.

Problemi

9-1 Limiti inferiori sull'ordinamento per confronti nel caso medio

In questo problema, si proverà un limite inferiore $\Omega(n \lg n)$ sul tempo di esecuzione atteso di qualunque algoritmo di ordinamento di n input che opera per confronti, sia deterministico che randomizzato. Si comincia con l'esaminare un ordinamento per confronti deterministico A con albero di decisione T_A. Si assuma che ogni permutazione degli input di A sia equamente probabile.

a. Si supponga che ogni foglia di T_A sia etichettata con il valore della probabilità di essere raggiunta dato un input casuale. Provare che sono etichettate con $1/n!$ esattamente $n!$ foglie e che le altre sono etichettate con 0.

b. Sia $D(T)$ la lunghezza di un cammino esterno di un albero T; cioè $D(T)$ sia la somma delle profondità di tutte le foglie di T. Sia T un albero con $k > 1$ foglie, e siano RT e LT i sottoalberi destro e sinistro di T. Mostrare che $D(T) = D(RT) + D(LT) + k$.

c. Sia $d(k)$ il valore minimo di $D(T)$ su tutti gli alberi di decisione T con $k > 1$ foglie. Mostrare che $d(k) = \min_{1 \le i \le k-1} \{d(i) + d(k-i) + k\}$. (*Suggerimento*: considerare un albero di decisione T con k foglie che raggiunge il minimo. Sia i il numero di foglie in LT e $k - i$ il numero di foglie in RT.)

d. Provare che per un dato valore di $k > 1$ e i nell'intervallo $1 \le i \le k-1$, la funzione $i \lg i + (k-i) \lg(k-i)$ è minimizzata per $i = k/2$. Concludere che $d(k) = \Omega(k \lg k)$.

e. Provare che, per T_A, $D(T_A) = \Omega(n! \lg(n!))$ e concludere che il tempo atteso per ordinare n elementi è $\Omega(n \lg n)$.

Si consideri adesso un ordinamento per confronti *randomizzato* B. Si può estendere il modello ad albero di decisione per gestire la "randomizzazione" prevedendo due tipi di nodi: nodi di confronto ordinario e nodi "randomizzati". Un nodo randomizzato rappresenta una scelta casuale della forma RANDOM(1, r) fatta dall'algoritmo B: il nodo ha r figli, ognuno scelto in modo equamente probabile durante un'esecuzione dell'algoritmo.

f. Mostrare che per qualunque ordinamento per confronti randomizzato B, esiste un ordinamento per confronti deterministico A che, in media, non esegue più confronti di quanti ne faccia l'algoritmo B.

9-2 Ordinamento in loco in tempo lineare

a. Si supponga di avere un array di n record di dati da ordinare e che la chiave di ogni record abbia valore 0 o 1. Dare un semplice algoritmo con tempo lineare che ordini in loco gli n record. Oltre alla memoria fornita dall'array non usare memoria aggiuntiva se non di dimensione costante.

b. L'ordinamento progettato per la parte (a) può essere usato dal radix sort su n record con chiavi grandi b bit in tempo $O(bn)$? Giustificare la risposta.

c. Si supponga che n record abbiano chiavi nell'intervallo da 1 a k. Si modifichi il counting sort in modo che i record possano essere ordinati in loco in tempo $O(n + k)$. Si può usare, oltre all'array di input, $O(k)$ memoria. (*Suggerimento*: come si potrebbe fare per $k = 3$?)

Note al capitolo

Il modello ad albero di decisione per studiare ordinamenti per confronti fu introdotto da Ford e Johnson [72]. L'ampia dissertazione di Knuth sull'ordinamento [123] comprende, oltre ai limiti inferiori basati sulla teoria dell'informazione, descritti in questo capitolo, molte varianti del problema dell'ordinamento. Limiti inferiori sugli ordinamenti che usano generalizzazioni del modello ad albero di decisione sono stati ampiamente studiati da BenOr [23].

Knuth attribuisce a H. H. Seward, sia la progettazione del counting sort, nel 1954, sia l'idea di combinare il counting sort con il radix sort. L'ordinamento radix sort rispetto alla cifra meno significativa sembra che fosse un algoritmo noto, già ampiamente usato da operatori di macchine meccaniche ordinatrici di schede. Secondo Knuth la prima pubblicazione su questo metodo si trova in un documento del 1929 di L. J. Comrie che descrive le macchine perforatrici di schede. Il bucket sort è stato usato sin dal 1956, quando l'idea base fu proposta da E. J. Isaac e R. C. Singleton.

Mediano e selezione

10

L'i-esima ***statistica d'ordine*** di un insieme di n elementi è l'i-esimo elemento più piccolo. Per esempio, il ***minimo*** di un insieme di elementi è la prima statistica d'ordine ($i = 1$), e il ***massimo*** è l'n-esima statistica d'ordine ($i = n$). Il ***mediano***, informalmente, è il "numero di mezzo" dell'insieme. Quando n è dispari, il mediano è unico, cadendo in $i = (n + 1)/2$. Quando n è pari, vi sono due mediani, che cadono in $i = n/2$ e in $i = n/2 + 1$. Quindi, qualunque sia la parità di n, i mediani cadono in $i = \lfloor (n + 1)/2 \rfloor$ e $i = \lceil (n + 1)/2 \rceil$. Per semplicità si preferisce evitare di considerare due mediani quando n è pari: in questo caso, si indicherà come unico mediano quello di indice inferiore $i = n/2$.

Questo capitolo affronta il problema di selezionare l'i-esimo elemento da un insieme di n elementi distinti. Si assume per comodità che l'insieme contenga numeri distinti, sebbene di fatto qualunque risultato potrà essere esteso alla situazione in cui l'insieme contenga valori ripetuti. Il problema della ***selezione*** può essere specificato formalmente come segue:

Input: un insieme A di n numeri (distinti) e un numero i, con $1 \le i \le n$.

Output: l'elemento $x \in A$ che è più grande di esattamente altri $i - 1$ elementi di A.

Il problema della selezione può essere risolto in tempo $O(n \lg n)$: infatti si possono ordinare i numeri usando l'heapsort o il merge sort e poi semplicemente indirizzare l'i-esimo elemento dell'array di output. Vi sono comunque algoritmi più efficienti.

Nel paragrafo 10.1, si esaminerà il problema di selezionare il minimo e il massimo di un insieme di elementi. Più interessante è il problema di selezione generale, che viene affrontato nei due successivi paragrafi. Il paragrafo 10.2 analizza un algoritmo pratico che raggiunge un limite $O(n)$ del tempo di esecuzione nel caso medio. Il paragrafo 10.3 contiene un algoritmo di maggior interesse teorico che raggiunge un tempo di esecuzione $O(n)$ nel caso peggiore.

10.1 Minimo e massimo

Quanti confronti sono necessari per determinare il minimo in un insieme di n elementi? Si può facilmente ottenere un limite superiore uguale a $n - 1$ confronti: si esamina ogni elemento dell'insieme, uno alla volta, e si tiene traccia dell'elemento più piccolo incontrato fino a quel momento. Nella procedura seguente, si assume che l'insieme risieda in un array A, dove $length[A] = n$.

```
MINIMUM(A)
min ← A[1]
for i ← 2 to length[A]
```

```
3      do if min > A [i]
4            then min ← A[i]
5   return min
```

Naturalmente, in modo analogo si può trovare anche il massimo con $n - 1$ confronti.

Ciò è quanto di meglio si possa fare? Sì, poiché, per il problema di determinazione del minimo, si può ottenere anche un limite inferiore di $n - 1$ confronti. Si pensi a un algoritmo per la determinazione del minimo come a un torneo tra gli elementi. Ogni confronto è una partita del torneo in cui il più piccolo dei due elementi vince. L'osservazione cruciale è che ogni elemento eccetto il vincitore deve perdere almeno una partita. Quindi, sono necessari $n - 1$ confronti per determinare il minimo, e l'algoritmo MINIMUM è ottimo rispetto al numero di confronti eseguiti.

Un interessante raffinamento dell'analisi è la valutazione del numero medio di volte che la linea 4 viene eseguita. Il Problema 6-2 richiede di mostrare che questo numero medio è $\Theta(\lg n)$.

Minimo e massimo simultanei

In alcune applicazioni, bisogna trovare sia il minimo che il massimo in un insieme di n elementi. Per esempio, un programma grafico può aver bisogno di rappresentare in scala un insieme di dati (x, y) da inserire all'interno di uno schermo rettangolare o di altri dispositivi grafici di output. Per far ciò il programma deve prima determinare il minimo e il massimo di ogni coordinata.

Non è troppo difficile ideare un algoritmo che possa trovare sia il minimo che il massimo di n elementi usando un numero di confronti $\Omega(n)$ asintoticamente ottimale. Basta semplicemente trovare il minimo e il massimo in modo indipendente, usando $n - 1$ confronti per ognuno, per un totale di $2n - 2$ confronti.

In realtà, sono sufficienti solo $3\lceil n/2 \rceil$ confronti per trovare simultaneamente il minimo e il massimo. Per far ciò, si mantengono gli elementi minimo e massimo via via incontrati, ma piuttosto che analizzare ogni elemento dell'input confrontandolo con il minimo e il massimo correnti, con un costo di due confronti per elemento, si analizzano gli elementi *in coppia*. Si confrontano i due elementi della coppia in input, quindi si confronta il più piccolo con il minimo corrente e il più grande con il massimo corrente, con un costo di tre confronti per coppia.

Esercizi

10.1-1 Mostrare che il secondo elemento più piccolo di n elementi può essere trovato con $n + \lceil \lg n \rceil - 2$ confronti nel caso peggiore. (*Suggerimento*: trovare anche l'elemento più piccolo.)

* *10.1-2* Mostrare che sono necessari $\lceil 3n/2 \rceil - 2$ confronti per trovare, nel caso peggiore, sia il minimo che il massimo di n numeri. (*Suggerimento*: considerare quanti numeri sono potenzialmente il massimo o il minimo e analizzare come un confronto influenzi questo calcolo.)

10.2 Selezione con tempo medio lineare

Il problema generale della selezione appare più difficile del semplice problema di trovare un minimo, eppure, sorprendentemente, per entrambi i problemi il tempo di esecuzione asintotico è lo stesso: $\Theta(n)$. In questo paragrafo, si presenterà un algoritmo divide-et-impera per il problema della selezione. L'algoritmo RANDOMIZED-SELECT segue lo stesso modello dell'algoritmo quicksort del Capitolo 8. Come per il quicksort l'idea è di partizionare ricorsivamente l'array di input, ma diversamente dal quicksort, che analizza ricorsivamente entrambi i lati della partizione, RANDOMIZED-SELECT lavora soltanto su un lato della partizione. Questa differenza risalta nell'analisi: laddove il quicksort ha un tempo medio di esecuzione di $\Theta(n \lg n)$, il tempo medio di RANDOMIZED-SELECT è $\Theta(n)$.

RANDOMIZED-SELECT usa la procedura RANDOMIZED-PARTITION introdotta nel paragrafo 8.3. Quindi, come RANDOMIZED-QUICKSORT, è un algoritmo randomizzato, poiché il suo comportamento è determinato in parte dall'output di un generatore di numeri casuali. Il seguente codice per RANDOMIZED-SELECT restituisce l'i-esimo elemento più piccolo dell'array $A[p \,.\,.\, r]$.

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p = r
2     then return A[p]
3  q ← RANDOMIZED-PARTITION(A, p, r)
4  k ← q − p + 1
5  if i ≤ k
6     then return RANDOMIZED-SELECT(A, p, q, i)
7     else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Dopo aver eseguito l'algoritmo RANDOMIZED-PARTITION nella linea 3, l'array $A[p \,.\,.\, r]$ è partizionato in due sottoarray non vuoti $A[p \,.\,.\, q]$ e $A[q + 1 \,.\,.\, r]$ tali che ogni elemento di $A[p \,.\,.\, q]$ è minore o uguale ad ogni elemento di $A[q + 1 \,.\,.\, r]$. La linea 4 dell'algoritmo calcola il numero k di elementi nel sottoarray $A[p \,.\,.\, q]$. L'algoritmo quindi determina in quale dei due sottoarray $A[p \,.\,.\, q]$ e $A[q + 1 \,.\,.\, r]$ si trova l'i-esimo elemento più piccolo. Se $i \le k$, allora l'elemento desiderato si trova sul lato basso della partizione e viene ricorsivamente selezionato dal sottoarray nella linea 6. Invece, se $i > k$, l'elemento desiderato si trova sul lato alto della partizione. Poiché si conoscono già k valori più piccoli dell'i-esimo elemento di $A[p \,.\,.\, r]$ – cioè gli elementi di $A[p \,.\,.\, q]$ – l'elemento desiderato è l'$(i - k)$-esimo elemento più piccolo di $A[q + 1 \,.\,.\, r]$, che viene cercato ricorsivamente nella linea 7.

Il tempo di esecuzione di RANDOMIZED-SELECT nel caso peggiore è $\Theta(n^2)$, anche per trovare il minimo, perché si potrebbe essere così sfortunati da partizionare sempre usando gli elementi più grandi rimasti. L'algoritmo funziona bene, però, nel caso medio e poiché è randomizzato, nessun particolare input provoca il comportamento del caso peggiore.

Si può ottenere un limite superiore $T(n)$ sul tempo medio richiesto dalla RANDOMIZED-SELECT, che opera su un array di input di n elementi, come segue. Si è osservato nel paragrafo 8.4 che l'algoritmo RANDOMIZED-PARTITION produce una partizione il cui lato basso ha 1 elemento con probabilità $2/n$ e i elementi con probabilità $1/n$ per $i = 2, 3, \ldots, n - 1$. Assumendo

che $T(n)$ sia monotona crescente, nel caso peggiore RANDOMIZED-SELECT è sempre sfortunata nel senso che l'i-esimo elemento deve essere determinato sul lato più grande della partizione. Così si ha la ricorrenza

$$\begin{aligned} T(n) &\le \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n) \\ &\le \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2\rceil}^{n-1} T(k)\right) + O(n) \\ &= \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} T(k) + O(n) . \end{aligned}$$

La seconda riga segue dalla prima perché $\max(1, n-1) = n-1$ e

$$\max(k, n-k) = \begin{cases} k & \text{se } k \ge \lceil n/2\rceil , \\ n-k & \text{se } k < \lceil n/2\rceil . \end{cases}$$

Se n è dispari, ogni termine $T(\lceil n/2\rceil), T(\lceil n/2\rceil + 1), \ldots, T(n-1)$ compare due volte nella sommatoria, mentre se n è pari, ogni termine $T(\lceil n/2\rceil + 1), T(\lceil n/2\rceil + 2), \ldots, T(n-1)$ compare due volte e il termine $T(\lceil n/2\rceil)$ compare una volta. In entrambi i casi, la sommatoria della prima riga è limitata superiormente dalla sommatoria della seconda. La terza riga segue dalla seconda poiché nel caso peggiore $T(n-1) = O(n^2)$, e così il termine $(1/n)T(n-1)$ può essere assorbito dal termine $O(n)$.

Si risolve la ricorrenza per sostituzione. Si assuma che $T(n) < cn$ per qualche costante c che soddisfi le condizioni iniziali della ricorrenza. Usando l'ipotesi induttiva, si ha

$$\begin{aligned} T(n) &\le \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} ck + O(n) \\ &= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2\rceil - 1} k\right) + O(n) \\ &= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\left\lceil\frac{n}{2}\right\rceil - 1\right)\left\lceil\frac{n}{2}\right\rceil\right) + O(n) \\ &\le c(n-1) - \frac{c}{n}\left(\frac{n}{2} - 1\right)\left(\frac{n}{2}\right) + O(n) \\ &= c\left(\frac{3}{4}n - \frac{1}{2}\right) + O(n) \\ &\le cn , \end{aligned}$$

poiché si può scegliere c sufficientemente grande in modo che $c(n/4 + 1/2)$ domini il termine $O(n)$.

Così l'i-esimo elemento di un insieme, per qualsiasi i, e in particolare il mediano, possono essere determinati in tempo lineare nel caso medio.

Esercizi

10.2-1 Scrivere una versione iterativa di RANDOMIZED-SELECT.

10.2-2 Si supponga di usare RANDOMIZED-SELECT per selezionare l'elemento minimo dell'array $A = \langle 3, 2, 9, 0, 7, 5, 4, 8, 6, 1\rangle$. Descrivere una sequenza di partizioni che provochi una prestazione come quella del caso peggiore dell'algoritmo.

10.2-3 Si ricorda che, in presenza di elementi uguali, la procedura RANDOMIZED-PARTITION partiziona il sottoarray $A[p\,..\,r]$ in due sottoarray non vuoti $A[p\,..\,q]$ e $A[q+1\,..\,r]$ tali che ogni elemento in $A[p\,..\,q]$ sia minore *o uguale* ad ogni elemento in $A[q+1\,..\,r]$. Se sono presenti elementi uguali, la procedura RANDOMIZED-SELECT funziona correttamente?

10.3 Selezione in tempo lineare nel caso peggiore

Si esaminerà ora un algoritmo di selezione il cui tempo di esecuzione è $O(n)$ nel caso peggiore. Come RANDOMIZED-SELECT, l'algoritmo SELECT trova l'elemento desiderato tramite partizionamenti ricorsivi dell'array di input. L'idea di base dell'algoritmo, però, è di *garantire* una buona suddivisione quando l'array viene partizionato. SELECT usa l'algoritmo di partizionamento deterministico PARTITION del quicksort (si veda il paragrafo 8.1), modificato in modo da prendere come parametro di input l'elemento perno attorno al quale partizionare.

L'algoritmo SELECT determina l'i-esimo elemento più piccolo di un array di input di n elementi eseguendo i seguenti passi.

1. Divide gli n elementi dell'array di input in $\lfloor n/5\rfloor$ gruppi di 5 elementi ciascuno e al più un gruppo costituito dai rimanenti n mod 5 elementi.
2. Trova il mediano di ognuno degli $\lceil n/5\rceil$ gruppi, ordinando con l'insertion sort gli elementi (che sono al più 5) di ogni gruppo.
3. Usa SELECT ricorsivamente per trovare il mediano x degli $\lceil n/5\rceil$ mediani trovati al passo 2.
4. Partiziona l'array di input attorno al mediano dei mediani x usando una versione modificata di PARTITION. Sia k il numero di elementi sul lato basso della partizione, così che $n-k$ sia il numero di elementi sul lato alto.
5. Usa SELECT ricorsivamente per trovare l'i-esimo elemento più piccolo sul lato basso se $i \le k$, oppure l'$(i-k)$-esimo elemento più piccolo sul lato alto se $i > k$.

Per analizzare il tempo di esecuzione di SELECT, si determina un limite inferiore sul numero di elementi più grandi dell'elemento partizionante x. La figura 10.1 è utile per visualizzare questo calcolo. Almeno metà dei mediani trovati al passo 2 sono maggiori o uguali dell'elemento mediano dei mediani x. Così almeno metà degli $\lceil n/5\rceil$ gruppi contribuisce con 3 elementi più grandi di x, eccetto quel gruppo che ha meno di 5 elementi se n non è divisibile per 5, e quel gruppo contenente x stesso. Detraendo questi due gruppi, segue che il numero di elementi più grandi di x è almeno

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil - 2\right) \ge \frac{3n}{10} - 6 .$$

Figura 10.1 *Analisi dell'algoritmo* SELECT. *Gli n elementi sono rappresentati da piccoli cerchi e ogni gruppo occupa una colonna. Il mediano di ogni gruppo è bianco e il mediano dei mediani è etichettato con x. Le frecce vanno dagli elementi più grandi ai più piccoli: da ciò si può vedere che 3 dei 5 elementi di ogni gruppo alla destra di x sono maggiori di x e 3 dei 5 elementi di ogni gruppo alla sinistra di x sono minori di x. Gli elementi di x sono su un fondo grigio.*

Analogamente, il numero di elementi minori di x è almeno $3n/10 - 6$. Perciò, nel caso peggiore, SELECT è chiamata ricorsivamente su al più $7n/10 + 6$ elementi al passo 5.

Si può ora sviluppare una ricorrenza per il tempo di esecuzione $T(n)$ nel caso peggiore dell'algoritmo SELECT. I passi 1, 2 e 4 richiedono tempo $O(n)$. (Il passo 2 consiste di $O(n)$ chiamate di insertion sort su insiemi di dimensione $O(1)$). Il passo 3 richiede tempo $T(\lceil n/5 \rceil)$ e il passo 5 richiede al più tempo $T(7n/10 + 6)$ assumendo che T sia monotona crescente. Si noti che $7n/10 + 6 < n$ per $n > 20$ e che qualunque input di al più 80 elementi richiede tempo $O(1)$. Si può quindi ottenere la ricorrenza

$$T(n) \le \begin{cases} \Theta(1) & \text{se } n \le 80 , \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{se } n > 80 . \end{cases}$$

Dimostriamo, per sostituzione, che il tempo di esecuzione è lineare. Si assuma che $T(n) \le cn$ per ogni $n > 80$ e qualche costante c. Sostituendo nel lato destro della ricorrenza si ottiene

$$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c(7n/10 + 6) + O(n) \\ &\le cn/5 + c + 7cn/10 + 6c + O(n) \\ &\le 9cn/10 + 7c + O(n) \\ &\le cn , \end{aligned}$$

poiché si può scegliere c sufficientemente grande in modo che $c(n/10 - 7)$ sia maggiore della funzione descritta dal termine $O(n)$ per ogni $n > 80$. Il tempo di esecuzione di SELECT nel caso peggiore è quindi lineare.

Come negli ordinamenti per confronti (si veda il paragrafo 9.1), SELECT e RANDOMIZED-SELECT determinano le informazioni sull'ordine relativo tra elementi solo attraverso confronti. Perciò, il comportamento lineare non risulta da ipotesi sull'input, come accade nel caso degli algoritmi di ordinamento del Capitolo 9. L'ordinamento richiede tempo $\Omega(n \lg n)$ nel modello che opera per confronti, anche in media (si veda il Problema 9-1), e perciò il metodo, presentato nell'introduzione di questo capitolo, che prevede l'ordinamento e l'indicizzazione dell'elemento, è asintoticamente inefficiente.

Esercizi

10.3-1 Nell'algoritmo SELECT, gli elementi in input sono divisi in gruppi di 5. L'algoritmo funzionerebbe in tempo lineare se i gruppi fossero di 7 elementi? Si mostri che SELECT non funziona in tempo lineare se i gruppi sono di 3 elementi.

10.3-2 Analizzare SELECT per mostrare che il numero di elementi maggiori del mediano dei mediani x e il numero di elementi minori di x è almeno $\lceil n/4 \rceil$ se $n \ge 38$.

10.3-3 Mostrare come è possibile fare in modo che il quicksort sia eseguito in tempo $O(n \lg n)$ nel caso peggiore.

* *10.3-4* Si supponga che un algoritmo usi solo confronti per trovare l'i-esimo elemento più piccolo in un insieme di n elementi. Mostrare che si possono trovare anche gli $i - 1$ elementi più piccoli e gli $n - i$ elementi più grandi senza eseguire nessun ulteriore confronto.

10.3-5 Sia dato un sottoprogramma "black-box" (cioè a scatola chiusa) per trovare il mediano, che richieda tempo lineare nel caso peggiore. Si dia un semplice algoritmo con tempo lineare che risolva il problema della selezione per qualunque i.

10.3-6 I k-esimi ***quantili*** di un insieme di n elementi sono i $k - 1$ elementi che dividono l'insieme ordinato in k insiemi della stessa dimensione (o al più con 1 elemento di differenza). Fornire un algoritmo con tempo $O(n \lg k)$ per elencare i k-esimi quantili di un insieme.

10.3-7 Descrivere un algoritmo con tempo $O(n)$ che, dato un insieme S di n numeri distinti e un intero positivo $k \le n$, determini i k numeri in S che sono più vicini al mediano di S.

10.3-8 Siano $X[1 \ldots n]$ e $Y[1 \ldots n]$ due array di n numeri, già ordinati. Fornire un algoritmo con tempo $O(\lg n)$ per trovare il mediano fra tutti i $2n$ elementi negli array X e Y.

10.3-9 Il professor Benzolo è consulente presso una compagnia petrolifera, che sta progettando un grosso oleodotto che va da est a ovest attraverso un campo di petrolio con n pozzi. Da ogni pozzo, un condotto laterale deve essere connesso direttamente al condotto principale lungo il cammino più breve (o nord o sud), come mostrato nella figura 10.2. Date le coordinate x e y dei pozzi, come si dovrebbe scegliere il luogo ottimale del condotto principale (quello che minimizza la lunghezza totale dei condotti laterali)? Mostrare che il luogo ottimale può essere determinato in tempo lineare.

Figura 10.2 *Si vuole determinare la posizione dell'oleodotto est-ovest che minimizzi la lunghezza totale dei condotti laterali nord-sud.*

Problemi

10-1 *La sequenza degli i numeri più grandi*

Dato un insieme di n numeri, si desidera trovare la sequenza dei più grandi i numeri ordinati, usando un algoritmo basato su confronti. Trovare l'algoritmo che realizza ognuno dei seguenti metodi con il miglior tempo di esecuzione asintotico nel caso peggiore: analizzare i tempi di esecuzione dei metodi in termini di i e di n.

a. Ordinare tutti i numeri ed elencare gli i più grandi.

b. Costruire con i numeri dati una coda con priorità e chiamare EXTRACT-MAX i volte.

c. Usare un algoritmo di selezione per trovare l'i-esimo numero più grande, quindi partizionare e ordinare gli i numeri più grandi.

10-2 *Mediano pesato*

Per n elementi distinti $x_1, x_2, \ldots, x_n$ con pesi positivi $w_1, w_2, \ldots, w_n$ tali che $\sum_{i=1}^{n} w_i = 1$, il ***mediano pesato*** è l'elemento x_k che soddisfa le seguenti disuguaglianze:

$$\sum_{x_i < x_k} w_i < \frac{1}{2} \quad ,$$

e

$$\sum_{x_i > x_k} w_i \le \frac{1}{2} \; .$$

a. Dedurre che il mediano di $x_1, x_2, \ldots, x_n$ è il mediano pesato degli x_i con pesi $w_i = 1/n$ per $i = 1, 2, \ldots, n$.

b. Mostrare come calcolare il mediano pesato di n elementi con un tempo $O(n \lg n)$ nel caso peggiore, usando l'ordinamento.

c. Mostrare come calcolare il mediano pesato in tempo $\Theta(n)$ nel caso peggiore, usando un algoritmo per trovare il mediano che abbia un tempo lineare come SELECT del paragrafo 10.3.

Il ***problema della collocazione dell'ufficio postale*** è definito come segue. Siano dati n punti $p_1, p_2, \ldots, p_n$ con pesi associati $w_1, w_2, \ldots, w_n$. Si desidera trovare un punto p (non necessariamente uno di quelli in input) che minimizzi la somma $\sum_{i=1}^{n} w_i d(p, p_i)$, dove $d(a, b)$ è la distanza tra due punti a e b.

d. Spiegare che il mediano pesato è un'ottima soluzione per il problema di trovare l'ufficio postale a una dimensione, in cui i punti sono semplicemente numeri reali e la distanza tra i punti a e b è $d(a, b) = |a - b|$.

e. Trovare la soluzione per il problema della collocazione dell'ufficio postale a 2 dimensioni, in cui i punti sono le coppie di coordinate (x, y) e la distanza tra i punti $a = (x_1, y_1)$ e $b = (x_2, y_2)$ è la ***distanza Manhattan***: $d(a, b) = |x_1 - x_2| + |y_1 - y_2|$.

10-3 *Selezione di elementi piccoli*

Il numero $T(n)$ di confronti usati da SELECT per selezionare l'i-esimo elemento tra n numeri soddisfa, come è stato mostrato, $T(n) = \Theta(n)$, ma la costante nascosta dalla notazione Θ è piuttosto grande. Quando i è piccolo relativamente a n, si può realizzare una diversa procedura che usi SELECT come un sottoprogramma, ma esegua meno confronti nel caso peggiore.

a. Descrivere un algoritmo che usi $U_i(n)$ confronti per trovare l'i-esimo elemento più piccolo fra n elementi, dove $i \le n/2$ e

$$U_i(n) = \begin{cases} T(n) & \text{se } n \le 2i \; , \\ n/2 + U_i(n/2) + T(2i) & \text{altrimenti} \; . \end{cases}$$

(*Suggerimento*: si cominci con $\lfloor n/2 \rfloor$ confronti disgiunti tra coppie, e si riapplichi ricorsivamente sull'insieme contenente il più piccolo elemento di ogni coppia.)

b. Mostrare che $U_i(n) = n + O(T(2i) \lg(n/i))$.

c. Mostrare che se i è una costante, allora $U_i(n) = n + O(\lg n)$.

d. Mostrare che se $i = n/k$ per $k \ge 2$, allora $U_i(n) = n + O(T(2n/k) \lg k)$.

Note al capitolo

L'algoritmo per trovare il mediano in tempo lineare nel caso peggiore fu progettato da Blum, Floyd, Pratt, Rivest e Tarjan [29]. La versione con tempo medio veloce è dovuta a Hoare [97]. Floyd e Rivest [70] hanno sviluppato una versione con tempo medio anche migliore che partiziona attorno a un elemento selezionato ricorsivamente da un piccolo campione degli elementi.

III Strutture di dati

Introduzione

Gli insiemi sono fondamentali per gli informatici quanto lo sono per i matematici. Mentre gli insiemi matematici sono immutabili, gli insiemi manipolati dagli algoritmi possono, nel tempo, crescere, contrarsi o cambiare in altri modi. Tali insiemi saranno chiamati ***dinamici***. I prossimi cinque capitoli presentano alcune tecniche di base per rappresentare e manipolare sul calcolatore insiemi dinamici finiti.

Gli algoritmi possono richiedere alcuni diversi tipi di operazioni da eseguire sugli insiemi. Per esempio, molti algoritmi necessitano solo della possibilità di *inserire* e *cancellare* elementi e *verificare l'appartenenza* a un insieme. Un insieme dinamico che offre queste operazioni è chiamato un ***dizionario***. Altri algoritmi richiedono operazioni più complicate. Per esempio, le code con priorità, introdotte nel Capitolo 7 nel contesto della struttura di dati heap, offrono le operazioni di inserzione di un elemento e di estrazione del più piccolo elemento. È chiaro quindi che il miglior modo di realizzare un insieme dinamico dipende dalle operazioni che devono essere fornite.

Elementi di un insieme dinamico

In una tipica realizzazione di un insieme dinamico, ogni elemento è rappresentato da un oggetto i cui campi possono essere esaminati e manipolati se si ha un puntatore all'oggetto. (Il Capitolo 11 discute la realizzazione di oggetti e puntatori in ambienti di programmazione che non li contengono come tipi di dato primitivi.) Alcuni insiemi dinamici assumono che uno dei campi dell'oggetto sia un campo ***chiave*** che lo identifica. Se le chiavi sono tutte diverse, si può pensare agli insiemi dinamici come a un insieme di valori chiave. L'oggetto può contenere ***dati satellite***, che occupano altri campi dell'oggetto ma non sono utilizzati in nessun modo nella realizzazione dell'insieme; può avere anche campi che sono manipolati dalle operazioni su insiemi: questi campi possono contenere dati o puntatori ad altri oggetti nell'insieme.

Alcuni insiemi dinamici presuppongono che le chiavi siano tratte da un insieme totalmente ordinato, come i numeri reali, o l'insieme di tutte le parole secondo l'usuale ordinamento alfabetico. (Un insieme totalmente ordinato soddisfa la proprietà di tricotomia, definita nel Capitolo 2.) Un ordinamento totale permette di definire il minimo elemento dell'insieme, per esempio, o di parlare del prossimo elemento più grande di un dato elemento in un insieme.

Operazioni su insiemi dinamici

Le operazioni su insiemi dinamici possono essere raggruppate in due categorie: ***interrogazioni***, che restituiscono semplicemente informazioni sull'insieme e ***operazioni di modifica***, che modificano l'insieme. Segue una lista delle operazioni tipiche. Qualunque specifica applicazione di solito richiede che solo poche di queste vengano realizzate.

SEARCH(S, k) Un'interrogazione che, dato un insieme S e un valore chiave k, restituisce un puntatore x a un elemento in S tale che $key[x] = k$, oppure NIL se un tale elemento non appartiene ad S.

INSERT(S, x) Un'operazione di modifica che inserisce nell'insieme S l'elemento puntato da x. Di solito si assume che tutti i campi nell'elemento x necessari alla realizzazione dell'insieme siano già stati inizializzati.

DELETE(S, x) Un'operazione di modifica che, dato un puntatore x ad un elemento nell'insieme S, toglie x da S. (Si noti che questa operazione usa un puntatore a un elemento x, non un valore chiave.)

MINIMUM (S) Un'interrogazione su un insieme totalmente ordinato S che restituisce l'elemento di S con la chiave più piccola.

MAXIMUM (S) Un'interrogazione su un insieme totalmente ordinato S che restituisce l'elemento di S con la chiave più grande.

SUCCESSOR(S, x) Un'interrogazione che, dato un elemento x la cui chiave appartiene a un insieme totalmente ordinato S, restituisce il successivo elemento più grande in S, o NIL se x è l'elemento massimo.

PREDECESSOR(S, x) Un'interrogazione che, dato un elemento x la cui chiave appartiene a un insieme totalmente ordinato S, restituisce il successivo elemento più piccolo in S, o NIL se x è l'elemento minimo.

Le interrogazioni SUCCESSOR e PREDECESSOR sono spesso estese agli insiemi con chiavi non distinte. Per un insieme di n chiavi, una chiamata MINIMUM seguita da $n - 1$ chiamate a SUCCESSOR elenca gli elementi dell'insieme in modo ordinato.

Il tempo per eseguire un'operazione su insiemi è di solito misurato in termini della cardinalità dell'insieme, che è fornita come argomento. Per esempio, il Capitolo 14 descrive una struttura di dati che può fornire una qualunque delle operazioni descritte precedentemente su un insieme di cardinalità n in tempo $O(\lg n)$.

Sommario della III parte

I capitoli dall'11 al 15 descrivono alcune strutture di dati che possono essere usate per realizzare insiemi dinamici; molte di queste saranno usate in seguito per costruire algoritmi efficienti per svariati problemi. Un'altra importante struttura – lo heap – è stata presentata nel Capitolo 7.

N.d.R. Notiamo che il linguaggio usato per lo pseudocodice non dispone di una notazione per distinguere tra un *puntatore* a un oggetto e l'*oggetto* stesso. L'interpretazione corretta è facilmente ricavata dal contesto. Per esempio, poiché il passaggio dei parametri a una procedura avviene *per valore* (si veda il paragrafo 1.1), se una procedura deve modificare un oggetto (come la DELETE qui di seguito), occorrerà passare alla procedura un *puntatore* x all'oggetto. Viceversa, se si fa riferimento a un attributo o campo di un oggetto x, x denoterà l'*oggetto*, in quel contesto.

Il Capitolo 11 presenta i concetti essenziali per lavorare con strutture di dati semplici come pile, code, liste concatenate e alberi radicati. Viene anche mostrato come oggetti e puntatori possano essere realizzati in ambienti di programmazione che non li forniscono come primitivi. Molto di questo materiale dovrebbe essere familiare a chiunque abbia già seguito un corso di introduzione alla programmazione.

Il Capitolo 12 presenta le tabelle hash, che forniscono le operazioni dei dizionari INSERT, DELETE e SEARCH. Nel caso peggiore, è richiesto un tempo $\Theta(n)$ per eseguire un'operazione di SEARCH, ma il tempo medio per le operazioni sulle tabelle hash è di $O(1)$. L'analisi delle tabelle hash si basa sulla probabilità, ma la maggior parte del capitolo non richiede preparazione sull'argomento.

Gli alberi binari di ricerca, che sono affrontati nel Capitolo 13, forniscono tutte le operazioni sugli insiemi dinamici elencate sopra. Nel caso peggiore ogni operazione richiede tempo $\Theta(n)$ su un albero con n elementi, ma su un albero binario di ricerca costruito in modo casuale il tempo medio per ogni operazione è $O(\lg n)$. Gli alberi binari di ricerca servono come base per molte altre strutture di dati.

Gli alberi RB, una variante degli alberi binari di ricerca, sono introdotti nel Capitolo 14. Diversamente dagli alberi di ricerca binari, gli alberi RB offrono sempre buone prestazioni: le operazioni richiedono un tempo $O(\lg n)$ nel caso peggiore. Un albero RB è un albero di ricerca *bilanciato*; il Capitolo 19 presenta un altro tipo di albero di ricerca bilanciato, chiamato B-albero. Benché i meccanismi degli alberi RB siano abbastanza complicati, si possono cogliere molte delle loro proprietà senza studiarne i meccanismi in dettaglio. Tuttavia, può essere molto istruttivo dare un'occhiata al codice dei programmi presentati nel capitolo.

Nel Capitolo 15, si mostra come estendere gli alberi RB per offrire operazioni diverse da quelle di base elencate precedentemente. Prima si estendono per mantenere dinamicamente l'i-esimo elemento di un insieme di chiavi, quindi si estendono in modo diverso per mantenere intervalli di numeri reali.

Strutture di dati fondamentali

11

In questo capitolo si esaminerà la rappresentazione di insiemi dinamici attraverso semplici strutture di dati che usano puntatori. Sebbene molte strutture di dati complesse possano essere modellate usando puntatori, saranno presentate solo quelle più semplici: pile, code, liste concatenate e alberi radicati. Si discuterà anche un metodo per rappresentare oggetti e puntatori con gli array.

11.1 Pile e Code

Pile e code sono insiemi dinamici in cui l'elemento rimosso dall'insieme con l'operazione DELETE è prestabilito. In una ***pila***, l'elemento cancellato dall'insieme è quello inserito più di recente: la pila realizza una politica ***ultimo arrivato, primo servito***, detta *LIFO* (last-in, first-out). Similmente, in una ***coda***, l'elemento cancellato è sempre quello che è stato nell'insieme più a lungo: la coda realizza una politica ***primo arrivato, primo servito***, detta *FIFO* (first-in, first-out). Vi sono diversi modi efficienti per realizzare pile e code su un calcolatore. In questo paragrafo si mostrerà come realizzarle con un semplice array.

Pile

L'operazione di INSERT su una pila è spesso chiamata PUSH e l'operazione DELETE, che non ha nessun elemento come argomento, è spesso chiamata POP. Questi termini inglesi sono quelli usati per descrivere l'utilizzo delle pile di piatti caricate a molla, che si trovano nelle caffetterie americane. L'ordine in cui i piatti sono estratti dalla pila è l'ordine contrario a quello con cui sono stati inseriti, infatti solo il piatto in cima è accessibile.

Come mostrato nella figura 11.1, si può realizzare una pila di al più n elementi con un array $S[1 \ldots n]$. L'array ha un attributo $top[S]$ che è l'indice dell'elemento inserito più di recente. La pila è costituita dagli elementi $S[1 \ldots top[S]]$, dove $S[1]$ è l'elemento in fondo alla pila e $S[top[S]]$ è l'elemento in cima.

Quando $top[S] = 0$, la pila non contiene alcun elemento ed è ***vuota***. Si può controllare se la pila è vuota con l'operazione di interrogazione STACK-EMPTY. Se si esegue un'operazione POP su una pila vuota, si dice che la pila va in ***underflow***, che ovviamente è un errore. Se $top[S]$ supera n, la pila va in ***overflow***. (Nella realizzazione con lo pseudocodice, non ci si occuperà dell'overflow della pila).

Figura 11.1 *La realizzazione di una pila S con un array. Gli elementi della pila appaiono solo nelle posizioni grigie chiare.* (a) *La pila S ha 4 elementi. L'elemento in cima è* 9. (b) *La pila S dopo le chiamate* PUSH(*S*, 17) *e* PUSH (*S*, 3). (c) *La pila S dopo la chiamata* POP(*S*) *ha restituito l'elemento* 3, *che è quello inserito più recentemente. Sebbene l'elemento* 3 *appaia ancora nell'array, esso non è più nella pila; l'elemento in cima è il* 17.

Ognuna delle operazioni sulla pila può essere realizzata con poche linee di codice.

```
STACK-EMPTY(S)
  if top[S] = 0
      then return TRUE
      else return FALSE

PUSH(S, x)
  top[S] ← top[S] + 1
  S[top[S]] ← x

POP(S)
  if STACK-EMPTY(S)
      then error "underflow"
      else top[S] ← top[S] − 1
           return S[top[S] + 1]
```

La figura 11.1 mostra gli effetti delle operazioni di modifica PUSH e POP. Ognuna delle tre operazioni sulla pila richiede tempo $O(1)$.

Code

L'operazione di INSERT su una coda è chiamata ENQUEUE e l'operazione di DELETE è chiamata DEQUEUE; come l'operazione POP della pila, DEQUEUE non ha alcun elemento come argomento. La proprietà FIFO fa sì che una coda operi come una fila di persone in un ufficio pubblico. La coda ha una *testa* (head) e una *coda* (tail). Quando un elemento viene inserito, prende posto in fondo alla coda, come una persona appena arrivata prende posto alla fine della fila. L'elemento estratto è sempre quello in testa alla coda, come la persona in testa alla fila che sta aspettando da più tempo. (Fortunatamente, non bisogna preoccuparsi di elementi della fila che non rispettano il proprio turno.)

La figura 11.2 mostra un modo di realizzare una coda di al più $n-1$ elementi usando un array $Q[1 \ldots n]$. La coda ha un attributo $head[Q]$ che è l'indice della testa, o, come si dice, che punta alla testa. L'attributo $tail[Q]$ è l'indice della prossima locazione in cui l'ultimo arrivato sarà inserito nella coda. Gli elementi della coda sono alle locazioni $head[Q]$, $head[Q]+1, \ldots, tail$

Figura 11.2 *La realizzazione di una coda con un array* $Q[1 \ldots 12]$. *Gli elementi della coda appaiono solo nelle posizioni grigie chiare.* (a) *La coda Q ha* 5 *elementi in* $Q[7 \ldots 11]$. (b) *La configurazione della coda dopo le chiamate* ENQUEUE(*Q*, 17), ENQUEUE(*Q*, 3) *e* ENQUEUE (*Q*, 5). (c) *La configurazione della coda dopo la chiamata* DEQUEUE(*Q*) *restituisce il valore* 15 *della chiave inizialmente alla testa della coda. La nuova testa ha chiave* 6.

$[Q]-1$ e si procede in modo *circolare*, nel senso che la locazione 1 segue immediatamente la locazione n secondo un ordine circolare. Quando $head[Q] = tail[Q]$ la coda è vuota. Inizialmente si ha $head[Q] = tail[Q] = 1$. Quando la coda è vuota e si tenta di estrarre un elemento la coda va in underflow. Quando $head[Q] = tail[Q] + 1$ la coda è piena e se si tenta di inserire un elemento la coda va in overflow.

Nelle seguenti procedure ENQUEUE e DEQUEUE, i controlli dell'errore di overflow e di underflow sono stati omessi. (L'Esercizio 11.1-4 richiede di fornire un codice che controlli queste due condizioni di errore.)

```
ENQUEUE (Q, x)
  Q[tail[Q]] ← x
  if tail[Q] = length[Q]
      then tail[Q] ← 1
      else tail[Q] ← tail[Q] + 1

DEQUEUE (Q)
  x ← Q[head [Q]]
  if head [Q] = head [Q]
      then head [Q] ← 1
      else head [Q] ← head [Q] + 1
  return x
```

La figura 11.2 mostra gli effetti delle operazioni ENQUEUE e DEQUEUE. Ogni operazione richiede un tempo $O(1)$.

Esercizi

11.1-1 Usando la figura 11.1 come modello, illustrare il risultato di ognuna delle seguenti operazioni: PUSH(S, 4), PUSH(S, 1), PUSH (S, 3), POP(S), PUSH (S, 8) e POP(S), applicate ad una pila S inizialmente vuota memorizzata nell'array $S[1 \ldots 6]$.

11.1-2 Spiegare come realizzare due pile con un array $A[1 \ldots n]$ in modo tale che nessuna di esse vada in overflow a meno che il numero complessivo di elementi sia n. Le operazioni di POP e PUSH dovrebbero richiedere tempo $O(1)$.

11.1-3 Usando la figura 11.2 come modello, illustrare il risultato di ognuna delle seguenti operazioni: ENQUEUE(Q, 4), ENQUEUE(Q, 1), ENQUEUE(Q, 3), DEQUEUE(Q), ENQUEUE(Q, 8) e DEQUEUE(Q), applicate ad una coda Q inizialmente vuota memorizzata nell'array $Q[1 \ldots 6]$.

11.1-4 Riscrivere ENQUEUE e DEQUEUE per controllare l'overflow e l'underflow di una coda.

11.1-5 Mentre una pila consente l'inserzione e la cancellazione di elementi da una sola parte e una coda consente l'inserzione ad un estremo e la cancellazione dall'altro, una ***deque*** (doppia coda) permette l'inserzione e la cancellazione da entrambi gli estremi. Scrivere quattro procedure con tempo $O(1)$ per inserire e cancellare elementi da entrambi gli estremi di una deque realizzata con un array.

11.1-6 Mostrare come realizzare una coda usando due pile. Analizzare il tempo di esecuzione delle operazioni della coda.

11.1-7 Mostrare come realizzare una pila usando due code. Analizzare il tempo di esecuzione delle operazioni della pila.

11.2 Liste concatenate

Una ***lista concatenata*** è una struttura di dati in cui gli oggetti sono sistemati secondo un ordine lineare. Diversamente da un array però, in cui l'ordine lineare è determinato dagli indici dell'array, l'ordine in una lista concatenata è determinato attraverso un puntatore ad ogni oggetto. Le liste concatenate forniscono una rappresentazione semplice e flessibile degli insiemi dinamici, offrendo tutte le operazioni già elencate per gli insiemi dinamici, anche se non necessariamente nel modo più efficiente.

Come mostrato nella figura 11.3 ogni elemento di una ***lista concatenata doppia*** o ***lista bidirezionale*** L è un oggetto con un campo *chiave* e altri due campi puntatore: *next* e *prev*. L'oggetto può contenere anche altri dati satellite. Dato un elemento x nella lista, *next*[x] punta al suo successore nella lista, mentre *prev*[x] punta al suo predecessore. Se *prev*[x] = NIL, l'elemento x non ha predecessore ed è quindi il primo elemento, o *testa*, della lista. Se *next*[x] = NIL, l'elemento x non ha successore ed è quindi l'ultimo elemento, o ***coda***, della lista. Un attributo *head*[L] punta al primo elemento della lista. Se *head*[L] = NIL la lista è vuota.

Figura 11.3 (a) *Una lista L bidirezionale che rappresenta l'insieme dinamico* {1, 4, 9, 16}. *Ogni elemento nella lista è un oggetto con campi per la chiave e i puntatori (rappresentati con frecce) all'oggetto successivo e precedente. Il campo next della coda e il campo prev della testa contengono* NIL, *indicati con una barra diagonale. L'attributo head* [L] *punta alla testa.* (b) *Dopo l'esecuzione di* LIST-INSERT(L, x), *dove key*[x] = 25 *la lista concatenata ha un nuovo oggetto con chiave* 25 *come nuova testa. Questo nuovo oggetto punta alla vecchia testa con chiave* 9. (c) *Il risultato della successiva chiamata* LIST-DELETE(L, x), *dove x punta all'oggetto con chiave* 4.

Una lista può avere diverse forme. Può essere concatenata semplice o bidirezionale, può essere o no ordinata, può essere o no circolare. Se una lista è ***concatenata semplice***, si omette il puntatore *prev* di ogni elemento. Se una lista è ***ordinata***, l'ordine lineare della lista corrisponde all'ordine lineare delle chiavi memorizzate negli elementi della lista; l'elemento minimo è la testa della lista mentre il massimo è la coda. Se la lista è ***non ordinata*** gli elementi possono apparire in qualsiasi ordine. In una ***lista circolare***, il puntatore *prev* della testa della lista punta alla coda e il puntatore *next* della coda della lista punta alla testa. La lista può così essere vista come un anello di elementi. Nel resto di questo paragrafo, si assumerà che le liste concatenate con cui si lavora siano non ordinate e bidirezionali.

Ricerca in una lista concatenata

La procedura LIST-SEARCH(L, k) trova il primo elemento con chiave k nella lista L con una semplice ricerca lineare, restituendo il puntatore all'elemento. Se nessun oggetto con chiave k appare nella lista, allora viene restituito NIL. Per la lista concatenata nella figura 11.3(a), la chiamata LIST-SEARCH(L, 4) restituisce un puntatore al terzo elemento e la chiamata LIST-SEARCH(L, 7) restituisce NIL.

```
LIST-SEARCH (L, k)
1  x ← head[L]
2  while x ≠ NIL e key [x] ≠ k
3      do x ← next[x]
4  return x
```

Per la ricerca in una lista di n oggetti, la procedura LIST-SEARCH richiede tempo $\Theta(n)$ nel caso peggiore, infatti la ricerca potrebbe richiedere l'esame dell'intera lista.

Inserimento in una lista concatenata

Dato un elemento x il cui campo chiave sia già stato inizializzato, la procedura LIST-INSERT inserisce x all'inizio della lista concatenata, come mostrato nella figura 11.3(b).

```
LIST-INSERT(L, x)
  next[x] ← head[L]
  if head[L] ≠ NIL
      then prev[head[L]] ← x
  head[L] ← x
  prev[x] ← NIL
```

Il tempo di esecuzione per LIST-INSERT su una lista di n elementi è $O(1)$.

Cancellazione da una lista concatenata

La procedura LIST-DELETE rimuove un elemento x da una lista concatenata L. La procedura riceve un puntatore a x e estrae l'elemento x dalla lista aggiornando i puntatori. Se si desidera cancellare un elemento con una data chiave, si deve prima chiamare LIST-SEARCH per recuperare il puntatore all'elemento.

```
LIST-DELETE(L, x)
  if prev[x] ≠ NIL
      then next[prev[x]] ← next[x]
      else head[L] ← next[x]
  if next[x] ≠ NIL
      then prev[next[x]] ← prev[x]
```

La figura 11.3(c) mostra come viene cancellato un elemento da una lista concatenata. LIST-DELETE viene eseguita in tempo $O(1)$, ma, se si desidera cancellare un elemento con una data chiave, è richiesto tempo $\Theta(n)$ nel caso peggiore perché si deve prima chiamare LIST-SEARCH.

Sentinelle

Il codice di LIST-DELETE potrebbe essere più semplice se si potesse evitare la gestione dei casi limite relativi alla testa e alla coda della lista.

```
LIST-DELETE'(L, x)
  next[prev[x]] ← next[x]
  prev[next[x]] ← prev[x]
```

Una ***sentinella*** è un oggetto fittizio che consente di semplificare la gestione dei casi limite. Per esempio, si supponga di fornire nella lista L un oggetto $nil[L]$ che rappresenti NIL, ma che abbia tutti i campi degli altri elementi della lista. Ovunque si abbia un riferimento a NIL nel codice, lo si sostituisce con un riferimento alla sentinella $nil[L]$. Come mostrato nella figura

Figura 11.4 *Una lista concatenata L che usa una sentinella $nil[L]$ (grigia scura) è la comune lista bidirezionale richiusa in modo circolare con $nil[L]$ disposto tra la testa e la coda. L'attributo $head[L]$ non è più necessario, poiché si può accedere alla testa della lista con $next[nil[L]]$.* (a) *Una lista vuota.* (b) *La lista concatenata della figura 11.3(a), con chiave 9 in testa e chiave 1 in coda.* (c) *La lista dopo l'esecuzione di* LIST-INSERT'(L, x) *dove $key[x] = 25$. Il nuovo oggetto diventa la testa della lista.* (d) *La lista dopo la cancellazione dell'oggetto con chiave 1. La nuova coda è l'oggetto con chiave 4.*

11.4, ciò trasforma una regolare lista bidirezionale in una lista circolare, con la sentinella $nil[L]$ posta tra la testa e la coda: il campo $next[nil[L]]$ punta alla testa della lista e $prev[nil[L]]$ punta alla coda. Analogamente, sia il campo $next$ della coda che il campo $prev$ della testa puntano a $nil[L]$. Poiché $next[nil[L]]$ punta alla testa, si può eliminare del tutto l'attributo $head[L]$, sostituendolo con il riferimento a $next[nil[L]]$. Una lista vuota consiste solo della sentinella, infatti sia $next[nil[L]]$ che $prev[nil[L]]$ possono essere inizializzati a $nil[L]$.

Il codice di LIST-SEARCH rimane lo stesso di prima, tranne che i riferimenti a NIL e $head[L]$ cambiano come specificato sopra.

```
LIST-SEARCH'(L, k)
  x ← next[nil[L]]
  while x ≠ nil[L] e key[x] ≠ k
      do x ← next[x]
  return x
```

Per cancellare un elemento dalla lista si usa la procedura LIST-DELETE'. Per inserire un elemento nella lista si usa la seguente procedura.

```
LIST-INSERT'(L, x)
  next[x] ← next[nil[L]]
  prev[next[nil[L]]] ← x
  next[nil[L]] ← x
  prev[x] ← nil[L]
```

La figura 11.4 mostra gli effetti di LIST-INSERT' e LIST-DELETE' su una lista campione.

Le sentinelle raramente riducono i limiti sui tempi asintotici delle operazioni sulle strutture di dati, ma possono ridurre i fattori costanti. Il vantaggio nell'uso delle sentinelle all'interno dei cicli riguarda di solito la chiarezza del codice piuttosto che la velocità: il codice delle liste concatenate, per esempio, è semplificato dall'uso delle sentinelle, ma si risparmia solo tempo $O(1)$ nelle procedure LIST-INSERT' e LIST-DELETE'. In altre situazioni, però, l'uso delle

sentinelle aiuta a contrarre il codice in un ciclo, riducendo così il coefficiente di n o n^2 (per esempio) nel tempo di esecuzione.

Le sentinelle non dovrebbero essere usate indiscriminatamente. Se vi sono molte piccole liste, la memoria extra usata per le sentinelle può rappresentare un significativo spreco di spazio. In questo libro, si useranno le sentinelle quando esse semplificano veramente il codice.

Esercizi

11.2-1 L'operazione INSERT su insiemi dinamici può essere realizzata su una lista concatenata semplice in tempo $O(1)$? Cosa si può dire per la DELETE?

11.2-2 Realizzare una pila usando una lista concatenata semplice L. Le operazioni PUSH e POP, dovrebbero richiedere ancora tempo $O(1)$.

11.2-3 Realizzare una coda usando una lista concatenata semplice L. Le operazioni ENQUEUE e DEQUEUE dovrebbero richiedere ancora tempo $O(1)$.

11.2-4 Realizzare le operazioni INSERT, DELETE e SEARCH dei dizionari usando liste circolari concatenate semplici. Quali sono i tempi di esecuzione delle procedure?

11.2-5 L'operazione UNION su insiemi dinamici prende come input due insiemi disgiunti S_1 e S_2 e restituisce un insieme $S = S_1 \cup S_2$ consistente di tutti gli elementi di S_1 e di S_2. Gli insiemi S_1 e S_2 sono di solito distrutti dall'operazione. Mostrare come fornire UNION con tempo $O(1)$ usando una struttura di dati a lista adeguata.

11.2-6 Scrivere una procedura che fonda due liste concatenate semplici ordinate in una singola lista concatenata semplice ordinata senza usare sentinelle. Poi, scrivere una procedura analoga usando una sentinella con chiave ∞ per marcare la fine di ogni lista. Confrontare le due procedure riguardo alla semplicità del codice.

11.2-7 Dare una procedura non ricorsiva con tempo $\Theta(n)$ che inverta una lista concatenata semplice di n elementi. La procedura dovrebbe usare solo una quantità costante di memoria oltre quella necessaria per la lista stessa.

* *11.2-8* Spiegare come realizzare liste bidirezionali usando solo un puntatore $np[x]$ per elemento invece dei soliti due (*next* e *prev*). Si assuma che tutti i puntatori possano essere interpretati come interi di k bit e definire $np[x]$ come l'"or esclusivo" tra i k bit di $next[x]$ e di $prev[x]$, cioè $np[x] = next[x]$ XOR $prev[x]$. (Il valore NIL è rappresentato con 0.) Si raccomanda di descrivere quale informazione è necessaria per accedere alla testa della lista. Mostrare come realizzare le operazioni SEARCH, INSERT e DELETE su tali liste. Mostrare inoltre come invertire una lista di questo tipo con un tempo $O(1)$.

Figura 11.5 *La lista concatenata della figura 11.3(a) rappresentata dagli array key, next e prev. Ogni striscia verticale degli array rappresenta un singolo oggetto. I puntatori memorizzati corrispondono agli indici dell'array mostrati in alto; le frecce mostrano come interpretarli. Le strisce grigie chiare contengono gli elementi della lista. La variabile L contiene l'indice della testa.*

11.3 Realizzazione di puntatori e oggetti

Come è possibile rappresentare puntatori e oggetti usando linguaggi, come il Fortran, che non li forniscono? In questo paragrafo, si vedranno due modi di realizzare strutture di dati concatenate senza un esplicito tipo di dato puntatore. Si realizzeranno oggetti con array e puntatori con indici degli array.

Rappresentazione di oggetti con array multipli

Si può rappresentare un insieme di oggetti che abbiano gli stessi campi usando un array per ogni campo. Per esempio, la figura 11.5 mostra come si può realizzare la lista concatenata della figura 11.3(a) con tre array. L'array *key* contiene i valori delle chiavi attualmente nell'insieme dinamico. I puntatori sono memorizzati negli array *next* e *prev*. Dato un indice x, $key[x]$, $next[x]$ e $prev[x]$ rappresentano un oggetto della lista concatenata. Secondo questa interpretazione, un puntatore x è semplicemente un indice comune agli array *key*, *next* e *prev*.

Nella figura 11.3(a), l'oggetto con chiave 4 segue l'oggetto con chiave 16 nella lista concatenata. Nella figura 11.5, la chiave 4 compare in $key[2]$, e la chiave 16 compare in $key[5]$, così si ha $next[5] = 2$ e $prev[2] = 5$. Sebbene la costante NIL compaia nel campo *next* della coda e nel campo *prev* della testa, di solito si usa un intero (come 0 o -1) che sicuramente non possa rappresentare un indice corrente nell'array. Una variabile L contiene l'indice L della testa della lista.

Nello pseudocodice, sono state usate le parentesi quadre per denotare sia l'accesso ad un array tramite indice che la selezione di un campo (attributo) di un oggetto. In entrambi i casi, il significato di $key[x]$, $next[x]$ e $prev[x]$ è quello che di solito viene attribuito anche nei linguaggi di programmazione effettivi.

Rappresentazione di oggetti con un solo array

Le parole nella memoria di un calcolatore sono tipicamente indirizzate con interi da 0 a $M-1$, dove M è grande in modo appropriato. In molti linguaggi di programmazione, un oggetto occupa un insieme contiguo di locazioni nella memoria del calcolatore. Un puntatore è semplicemente l'indirizzo della prima locazione di memoria dell'oggetto, e altre locazioni di memoria dentro l'oggetto possono essere indirizzate aggiungendo uno spostamento al puntatore.

Figura 11.6 *La lista concatenata delle figure* 11.3(*a*) *e* 11.5 *rappresentata con un array singolo A. Ogni elemento di lista è un oggetto che occupa un sottoarray contiguo di lunghezza* 3 *dentro l'array. I tre campi key, next e prev corrispondono rispettivamente agli spostamenti* 0, 1 *e* 2. *Un puntatore a un oggetto è l'indice del primo elemento dell'oggetto. Gli oggetti contenenti elementi della lista sono grigi chiari e le frecce mostrano la successione degli elementi della lista.*

Si può usare la stessa strategia per realizzare oggetti in ambienti di programmazione che non forniscono esplicitamente tipi di dato puntatore. Per esempio la figura 11.6 mostra come un singolo array A possa essere usato per memorizzare la lista concatenata delle figure 11.3(a) e 11.5. Un oggetto occupa un sottoarray contiguo $A[j \mathinner{..} k]$. Ogni campo dell'oggetto corrisponde a uno spostamento nell'intervallo tra 0 e $k - j$; l'indice j è il puntatore all'oggetto. Nella figura 11.6, gli spostamenti corrispondenti a *key*, *next* e *prev* sono rispettivamente 0, 1 e 2. Per leggere il valore di $prev[i]$, dato il puntatore i, si aggiunge al valore i del puntatore lo spostamento 2, leggendo quindi $A[i + 2]$.

La rappresentazione con un singolo array è flessibile nel senso che permette di memorizzare nello stesso array oggetti di lunghezza diversa. Il problema di gestire un tale insieme eterogeneo di oggetti è più difficile del problema di gestire un insieme omogeneo, dove tutti gli oggetti hanno gli stessi campi. Poiché molte delle strutture di dati che saranno considerate sono composte di elementi omogenei, sarà sufficiente usare la rappresentazione di oggetti con array multipli.

Allocazione e deallocazione di oggetti

Per inserire una chiave in un insieme dinamico rappresentato con una lista bidirezionale, si deve allocare un puntatore a un oggetto attualmente inutilizzato nella rappresentazione della lista concatenata. È pertanto conveniente gestire la memoria di oggetti non attualmente utilizzati nella rappresentazione della lista concatenata per potere allocare nuovi oggetti. In alcuni sistemi, è utilizzato un ***garbage collector*** per determinare quali oggetti sono inutilizzati. Molte applicazioni, però, sono abbastanza semplici e restituiscono direttamente gli oggetti inutilizzati al gestore della memoria. Si analizzerà ora il problema di allocare e liberare (o deallocare) oggetti omogenei usando l'esempio della lista bidirezionale rappresentata con array multipli.

Si supponga che gli array nella rappresentazione con array multipli abbiano lunghezza m e che in qualche istante l'insieme dinamico contenga $n \leq m$ elementi. Allora n oggetti rappresentano gli elementi attualmente nell'insieme dinamico, e i rimanenti $m - n$ oggetti sono ***liberi***; gli oggetti liberi possono essere usati per rappresentare elementi da inserire in futuro nell'insieme dinamico.

Si mantengono gli oggetti liberi in una singola lista concatenata, che si chiama ***lista libera***. La lista libera usa solo l'array *next*, che memorizza il puntatore successivo all'interno della

Figura 11.7 *Gli effetti delle procedure* ALLOCATE-OBJECT *e* FREE-OBJECT. (a) *La lista della figura* 11.5 *(grigia chiara) e una lista libera (grigia scura). Le frecce mostrano la successione degli elementi della lista libera.* (b) *Il risultato della chiamata* ALLOCATE-OBJECT() *(che restituisce l'indice* 4*), che pone in key*[4] *il valore* 25 *e chiama* LIST-INSERT(*L*, 4). *La nuova testa della lista libera è l'oggetto* 8, *che era contenuto in next*[4] *nella lista libera.* (c) *Dopo l'esecuzione di* LIST-DELETE (*L*, 5) *si chiama* FREE-OBJECT(5). *L'oggetto* 5 *diventa la nuova testa della lista libera e l'oggetto* 8 *lo segue nella lista libera.*

lista. La testa della lista libera è mantenuta in una variabile globale *free*. Quando l'insieme dinamico rappresentato dalla lista concatenata L non è vuoto, la lista libera può essere intrecciata con la lista L, come mostrato nella figura 11.7. Si noti che ogni oggetto nella rappresentazione è o nella lista L o nella lista libera, ma non in entrambe.

La lista libera è una pila: il prossimo oggetto allocato è quello liberato per ultimo. Si può usare una versione per le liste delle operazioni PUSH e POP per realizzare rispettivamente le procedure per allocare e liberare oggetti. Si supponga che la variabile globale *free* usata nelle seguenti procedure punti al primo elemento della lista libera.

```
ALLOCATE-OBJECT ()
if free = NIL
    then error "fine dello spazio libero"
    else  x ← free
          free ← next[x]
          return x

FREE-OBJECT(x)
next[x] ← free
free[x] ← x
```

La lista libera contiene inizialmente tutti gli n oggetti non allocati. Quando la lista libera è stata completamente utilizzata la procedura segnala un errore. Di solito si usa una sola lista libera per servire diverse liste concatenate. La figura 11.8 mostra due liste concatenate e una lista libera interconnesse negli array *key*, *next* e *prev*.

Figura 11.8 *Due liste concatenate, L_1 (grigia chiara) ed L_2 (grigia scura) e una lista libera (nera) interconnesse.*

Le due procedure vengono eseguite in tempo $O(1)$, che le rende interessanti da usare. Possono essere modificate per funzionare per qualunque insieme omogeneo di oggetti purché un qualunque campo dell'oggetto venga utilizzato come campo *next* nella lista libera.

Esercizi

11.3-1 Disegnare la sequenza $\langle 13, 4, 8, 19, 5, 11 \rangle$ memorizzata come una lista bidirezionale usando la rappresentazione su array multipli. Fare la stesso per la rappresentazione con un singolo array.

11.3-2 Scrivere le procedure ALLOCATE-OBJECT e FREE-OBJECT per un insieme omogeneo di oggetti rappresentati con un singolo array.

11.3-3 Perché non è necessario inizializzare i campi *prev* degli oggetti nella realizzazione delle procedure ALLOCATE-OBJECT e FREE-OBJECT?

11.3-4 Spesso si vogliono mantenere contigui nella memoria tutti gli elementi di una lista bidirezionale; allora si può usare, per esempio, la rappresentazione con array multipli impiegando le prime m locazioni. (È il caso di un ambiente di calcolo a memoria virtuale paginata.) Spiegare come le procedure ALLOCATE-OBJECT e FREE-OBJECT possano essere realizzate in modo che la rappresentazione sia compatta. Assumere che non vi siano puntatori a elementi della lista al di fuori della lista stessa. (*Suggerimento*: usare la realizzazione di una pila tramite array.)

11.3-5 Sia L una lista bidirezionale di lunghezza m memorizzata negli array *key*, *prev* e *next* di lunghezza n. Si supponga che questi array siano gestiti dalle procedure ALLOCATE-OBJECT e FREE-OBJECT che mantengono una lista libera F bidirezionale. Si supponga inoltre che degli n elementi, m siano nella lista L e $n - m$ siano nella lista libera. Scrivere una procedura COMPACTIFY-LIST(L, F) che, date la lista L e la lista libera F, sposti gli elementi di L in modo che occupino le posizioni 1, 2, ..., m dell'array e ricomponga correttamente la lista libera F, occupando le posizioni $m + 1$, $m + 2$, ..., n. La procedura dovrebbe avere tempo di esecuzione $\Theta(m)$ e dovrebbe usare solo una quantità costante di spazio extra. Dimostrare con cura la correttezza della procedura.

11.4 Rappresentazione di alberi radicati

I metodi per rappresentare le liste descritti nel precedente paragrafo si estendono a qualunque struttura di dati omogenea. In questo paragrafo, ci si interesserà in modo specifico al problema di rappresentare alberi radicati attraverso strutture di dati con puntatori. Prima si esaminano gli alberi binari, quindi si presenta un metodo per alberi radicati in cui i nodi possono avere un numero qualunque di figli.

Si rappresenta ogni nodo di un albero binario con un oggetto. Come per le liste concatenate, si assume che ogni nodo contenga un campo *key*. I restanti campi che interessano sono puntatori ad altri nodi e variano a seconda del tipo di albero.

Alberi binari

Come mostrato nella figura 11.9, si usano i campi *p*, *left* e *right* per memorizzare i puntatori al padre, al figlio sinistro e al figlio destro di ogni nodo di un albero binario T. Se $p[x] = \text{NIL}$, allora x è la radice. Se il nodo x non ha figlio sinistro, allora $left[x] = \text{NIL}$, e similmente per il figlio destro. Il puntatore alla radice dell'albero T è memorizzato nell'attributo $root[T]$. Se $root[T] = \text{NIL}$, l'albero è vuoto.

Alberi radicati con un numero illimitato di figli

Lo schema per rappresentare un albero binario può essere esteso a qualunque classe di alberi in cui il numero di figli di ogni nodo sia, al massimo, una qualche costante k: si sostituiscono i campi *left* e *right* con $child_1, child_2, \ldots, child_k$. Questo schema non funziona quando il numero di figli di un nodo è illimitato, poiché non si sa in anticipo quanti campi (cioè quanti array nella rappresentazione con array multipli) bisogna allocare. Inoltre, anche se il numero di figli è

Figura 11.9 *La rappresentazione di un albero binario T. Ciascun nodo x ha i seguenti campi: $p[x]$ (in alto), $left[x]$ (in basso a sinistra) e $right[x]$ (in basso a destra). I campi key non sono mostrati.*

Figura 11.10 *La rappresentazione figlio-sinistro, fratello-destro di un albero T. Ciascun nodo x ha i seguenti campi: p[x] (in alto), left-child[x] (in basso a sinistra) e right-sibling[x] (in basso a destra). Le chiavi non sono mostrate.*

limitato da una costante grande, quando molti nodi hanno pochi figli viene sprecata molta memoria.

Fortunatamente, vi è uno schema ingegnoso per rappresentare alberi con un numero arbitrario di figli usando alberi binari, che ha il vantaggio di usare solo uno spazio $O(n)$ per qualunque albero radicato con n nodi. La ***rappresentazione figlio-sinistro fratello-destro*** è mostrata nella figura 11.10. Come prima, ogni nodo contiene un puntatore p al padre e *root*[T] punta alla radice dell'albero T. Però, invece di avere un puntatore per ogni figlio, ogni nodo ha solo due puntatori:

1. *left-child*[x] punta al figlio più a sinistra del nodo x, e
2. *right-sibling*[x] punta al fratello immediatamente alla destra di x.

Se il nodo x non ha figli, allora *left-child*[x] = NIL e se il nodo x è il figlio più a destra di suo padre, allora *right-sibling*[x] = NIL.

Altre rappresentazioni degli alberi

Gli alberi radicati talvolta si rappresentano in altri modi. Nel Capitolo 7, per esempio, si è rappresentato uno heap, che si basa su un albero binario completo, tramite un singolo array e un indice. Gli alberi che compaiono nel Capitolo 22 vengono attraversati soltanto verso la radice: così sono presenti solo puntatori al padre e non vi sono puntatori ai figli. Sono possibili molti altri schemi. Lo schema migliore dipende dall'applicazione.

Esercizi

11.4-1 Disegnare l'albero binario, radicato nell'indice 6, che è rappresentato dai seguenti campi.

indice	*key*	*left*	*right*
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 Scrivere una procedura ricorsiva con tempo $O(n)$ che, dato un albero binario di n nodi, stampi le chiavi di ogni nodo dell'albero.

11.4-3 Scrivere una procedura non ricorsiva con tempo $O(n)$ che, dato un albero binario di n nodi, stampi le chiavi di ogni nodo dell'albero. Usare una pila come struttura di dati ausiliaria.

11.4-4 Scrivere una procedura con tempo $O(n)$ che stampi tutte le chiavi di un albero radicato arbitrario con n nodi, dove l'albero è memorizzato usando una rappresentazione figlio sinistro, fratello destro.

* *11.4-5* Scrivere una procedura non ricorsiva con tempo $O(n)$ che, dato un albero binario di n nodi, stampi la chiave di ogni nodo. Usare, oltre allo spazio per l'albero, una quantità costante di spazio extra e non modificare l'albero, neanche temporaneamente, durante la procedura.

* *11.4-6* La rappresentazione figlio sinistro, fratello destro di un albero radicato arbitrario usa tre puntatori in ogni nodo: *left-child*, *right-sibling* e *parent*. Da qualunque nodo, il padre e tutti i figli del nodo possono essere raggiunti. Mostrare come ottenere lo stesso risultato usando per ogni nodo solo due puntatori e un valore booleano.

Problemi

11-1 *Confronti tra liste*

Per ognuno dei quattro tipi di liste nella seguente tabella, qual è il tempo di esecuzione asintotico, nel caso peggiore, per ogni operazione elencata di seguito?

	non ordinata, concatenata semplice	ordinata, concatenata semplice	non ordinata, bidirezionale	ordinata, bidirezionale
SEARCH(L, k)				
INSERT(L, x)				
DELETE(L, x)				
SUCCESSOR(L, x)				
PREDECESSOR(L, x)				
MINIMUM(L)				
MAXIMUM(L)				

11-2 *Realizzazione di un mergeable heap con liste concatenate*

Un ***mergeable heap*** offre le seguenti operazioni: MAKE-HEAP (che crea un mergeable heap vuoto), INSERT, MINIMUM, EXTRACT-MIN e UNION. Mostrare come realizzare i mergeable heap usando liste concatenate in ognuno dei seguenti casi. Cercare le realizzazioni più efficienti possibili delle operazioni. Analizzare il tempo di esecuzione di ogni operazione in termini di cardinalità dell'insieme (o degli insiemi) dinamico su cui opera.

a. Le liste sono ordinate.

b. Le liste non sono ordinate.

c. Le liste non sono ordinate e gli insiemi dinamici da fondere sono disgiunti.

11-3 *Ricerca in una lista ordinata compatta*

L'Esercizio 11.3-4 richiede di mantenere in modo compatto una lista di n elementi nelle prime n posizioni di un array. Si assuma che tutte le chiavi siano distinte e che la lista compatta sia anche ordinata, cioè, key[i] < $key[next[i]]$ per ogni $i = 1, 2, \ldots, n$ tale che $next[i] \neq$ NIL. Sotto queste ipotesi, ci si aspetta che il seguente algoritmo randomizzato possa essere usato per la ricerca nella lista in un tempo $o(n)$.

```
COMPACT-LIST-SEARCH(L, k)
 1  i ← head[L]
 2  n ← length[L]
 3  while i ≠ NIL e key[i] ≤ k
 4      do j ← RANDOM(1, n)
 5          if key[i] < key[j] e key[j] < k
 6              then i ← j
 7          i ← next[i]
 8          if key[i] = k
 9              then return i
10  return NIL
```


Se si ignorano le linee 4-6 della procedura, si ha l'usuale algoritmo di ricerca in una lista concatenata ordinata, in cui l'indice i punta di volta in volta ad ogni posizione della lista. Le linee 4-6 cercano di avanzare ad una posizione j scelta a caso. Un tale avanzamento è vantaggioso se $key[j]$ è più grande di $key[i]$ e più piccolo di k; in tal caso j indica una posizione nella lista che i dovrebbe attraversare comunque durante l'usuale ricerca. Dato che la lista è compatta, si sa che qualunque scelta di j tra 1 ed n indica qualche oggetto della lista e non un elemento della lista libera.

a. Perché nella procedura COMPACT-LIST-SEARCH si ipotizza che tutte le chiavi siano distinte? Dedurre che quando la lista contiene chiavi ripetute non necessariamente gli avanzamenti casuali aiutano asintoticamente.

Si possono analizzare le prestazioni di COMPACT-LIST-SEARCH suddividendo la sua esecuzione in due fasi. Durante la prima fase, si trascura la ricerca di k che è eseguita nelle linee 7-9. Cioè la prima fase consiste soltanto dello spostamento in avanti nella lista attraverso avanzamenti casuali. Allo stesso modo, la fase 2 tralascia gli avanzamenti eseguiti nelle linee 4-6, e quindi opera come l'usuale ricerca lineare.

Sia X_t la variabile casuale che descrive la distanza nella lista concatenata (cioè, attraverso la catena dei puntatori *next*) dalla posizione i alla chiave desiderata k dopo t iterazioni della fase 1.

b. Dedurre che il tempo di esecuzione atteso di COMPACT-LIST-SEARCH è $O(t + E[X_t])$ per ogni $t \geq 0$.

c. Mostrare che $E[X_t] \leq \sum_{r=1}^{n} (1 - r/n)^t$. (*Suggerimento*: usare l'equazione (6.28).)

d. Mostrare che $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t+1)$.

e. Provare che $E[X_t] \leq n/(t+1)$ e spiegarne il significato intuitivo.

f. Mostrare che COMPACT-LIST-SEARCH è eseguita con tempo atteso $O(\sqrt{n})$.

Note al capitolo

Aho, Hopcroft e Ullman [5] e Knuth [121] sono ottimi riferimenti per le strutture di dati di base. Gonnet [90] fornisce dati sperimentali sulle prestazioni di molte operazioni sulle strutture di dati.

L'origine delle code e delle pile come strutture di dati in informatica è incerta, dato che nozioni corrispondenti già esistevano in matematica e nelle pratiche d'ufficio cartacee prima dell'introduzione dei calcolatori. Knuth [121] cita A. M. Turing per lo sviluppo di pile nei programmi di un "linker" nel 1947.

Anche per le strutture di dati basate sui puntatori non è noto un "inventore". Secondo Knuth i puntatori furono certamente usati nei primi calcolatori con le memorie a tamburo. Il linguaggio A-1 sviluppato da G. M. Hopper nel 1951 prevedeva la rappresentazione delle formule algebriche come alberi binari. Knuth dà credito al linguaggio IPL-II, sviluppato nel 1956 da A. Newell, J. C. Shaw e H. A. Simon, di aver riconosciuto l'importanza dei puntatori e di averne promosso l'utilizzo. Il loro linguaggio IPL-III sviluppato nel 1957 includeva esplicitamente operazioni sulle pile.

Tabelle hash

12

Molte applicazioni richiedono un'insieme dinamico che fornisca soltanto le operazioni INSERT, SEARCH, e DELETE dei dizionari. Per esempio, un compilatore di un linguaggio di programmazione mantiene una tabella dei simboli, in cui le chiavi degli elementi sono stringhe di caratteri qualunque che corrispondono a identificatori del linguaggio. Una tabella hash è una struttura di dati efficace per realizzare i dizionari. Sebbene la ricerca di un elemento in una tabella hash possa richiedere lo stesso tempo di ricerca in una lista concatenata – un tempo $\Theta(n)$ nel caso peggiore – in pratica le prestazioni del metodo hash sono estremamente buone. Sotto ragionevoli ipotesi, il tempo medio di ricerca di un elemento in una tabella hash è $O(1)$.

Una tabella hash è la generalizzazione del più semplice concetto di array ordinario. L'indirizzamento diretto di un array ordinario fa un uso efficace della possibilità di esaminare una posizione arbitraria di un array in un tempo $O(1)$. Il paragrafo 12.1 discute in dettaglio l'indirizzamento diretto. Esso può essere applicato quando ci si possa permettere di allocare un array con una posizione per ogni possibile chiave.

Quando il numero di chiavi effettivamente memorizzate è piccolo rispetto al numero totale di possibili chiavi, le tabelle hash diventano un'alternativa efficace all'indirizzamento diretto di un array, poiché una tabella hash usa tipicamente un array di dimensione proporzionale al numero di chiavi effettivamente memorizzate. Invece di usare la chiave come indice per indirizzare direttamente l'array, l'indice è *calcolato* utilizzando la chiave. Il paragrafo 12.2 presenta le idee principali e il paragrafo 12.3 descrive come gli indici dell'array possano essere calcolati a partire dalle chiavi utilizzando funzioni hash. Nel paragrafo inoltre sono presentate e analizzate alcune variazioni sul tema; la base comune a tutte è che l'uso delle tabelle hash è una tecnica estremamente efficace e pratica: le operazioni di base dei dizionari in media richiedono solo tempo $O(1)$.

12.1 Tabelle ad indirizzamento diretto

L'indirizzamento diretto è una tecnica semplice che funziona bene quando l'universo U delle chiavi è ragionevolmente piccolo. Si supponga che un'applicazione necessiti di un insieme dinamico in cui ogni elemento abbia una chiave ottenuta dall'universo $U = \{0, 1, \ldots, m-1\}$, dove m non sia troppo grande. Si assumerà che nessuna coppia di elementi abbia la stessa chiave.

Per rappresentare l'insieme dinamico, si usa un array, o ***tabella ad indirizzamento diretto***, $T[0 \ldots m-1]$, in cui ogni posizione, o *slot*, corrisponde ad una chiave nell'universo U. La

Figura 12.1 *La realizzazione di un insieme dinamico tramite una tabella ad indirizzamento diretto T. Ogni chiave nell'universo $U = \{0, 1, \ldots, 9\}$ corrisponde a un indice nella tabella. L'insieme di chiavi effettive determina le posizioni nella tabella che contengono i puntatori agli elementi. Le altre posizioni, grigie scure, contengono* NIL.

figura 12.1 illustra l'approccio; la posizione k punta ad un elemento dell'insieme con chiave k. Se l'insieme non contiene elementi con chiave k, allora $T[k] =$ NIL.

Le operazioni del dizionario sono banali da realizzare.

```
DIRECT-ADDRESS-SEARCH(T, k)
  return T[k]

DIRECT-ADDRESS-INSERT(T, x)
  T[key[x]] ← x

DIRECT-ADDRESS-Delete(T, x)
  T[key[x]] ← NIL
```

Ognuna di queste operazioni è veloce: è richiesto solo tempo $O(1)$.

Per alcune applicazioni, gli elementi nell'insieme dinamico possono essere memorizzati direttamente nella tabella ad indirizzamento diretto stessa. Cioè, piuttosto che memorizzare la chiave di un elemento e i suoi dati satellite in un oggetto esterno alla tabella, con un puntatore da una posizione della tabella all'oggetto, si può memorizzare l'oggetto nella posizione stessa, risparmiando così spazio. Inoltre spesso non è necessario memorizzare il campo chiave dell'oggetto, poiché se si ha l'indice di un oggetto nella tabella, si ha la sua chiave. Tuttavia, se una chiave non è memorizzata, si deve avere qualche modo per codificare che la posizione è vuota.

Esercizi

12.1-1 Si consideri un insieme dinamico S rappresentato da una tabella ad indirizzamento diretto T di lunghezza m. Descrivere una procedura che trovi il massimo elemento di S. Qual è il tempo di esecuzione della procedura nel caso peggiore?

12.1-2 Un ***vettore di bit*** è semplicemente un array di bit (0 e 1). Un vettore di bit di lunghezza m richiede molto meno spazio di un array di m puntatori. Descrivere come usare un vettore di bit per rappresentare un insieme dinamico di elementi distinti senza dati satellite. Le operazioni del dizionario dovrebbero richiedere tempo $O(1)$.

12.1-3 Suggerire come realizzare una tabella ad indirizzamento diretto in cui le chiavi memorizzate non siano necessariamente distinte e gli elementi possano avere dati satellite. Tutte le tre operazioni del dizionario (INSERT, DELETE e SEARCH) dovrebbero richiedere tempo $O(1)$. (Non si dimentichi che DELETE prende come argomento un puntatore all'oggetto da cancellare e non la sua chiave.)

* *12.1-4* Si desidera realizzare un dizionario usando l'indirizzamento in un array *grandissimo*. All'inizio gli elementi dell'array possono contenere dati non significativi e inizializzare l'intero array è impraticabile a causa della sua dimensione. Descrivere uno schema per realizzare un dizionario ad indirizzamento diretto su un array di questo tipo. Ogni oggetto memorizzato dovrebbe usare uno spazio $O(1)$; le operazioni INSERT, DELETE e SEARCH dovrebbero richiedere tempo $O(1)$ ognuna; l'inizializzazione della struttura di dati dovrebbe richiedere tempo $O(1)$. (*Suggerimento*: per aiutare a determinare se un dato elemento nell'array è valido o no, usare, come struttura addizionale, una pila la cui dimensione è data dal numero di chiavi effettivamente utilizzate nel dizionario.)

12.2 Tabelle hash

La difficoltà dell'indirizzamento diretto è ovvia: se l'universo U è grande, memorizzare una tabella T di dimensione $|U|$ può essere impraticabile, o addirittura impossibile, a causa della limitatezza della memoria disponibile su un calcolatore. Inoltre, l'insieme K di chiavi *effettivamente utilizzate* può essere così piccolo rispetto ad U che la maggior parte dello spazio allocato per T sarebbe inutilizzato.

Quando l'insieme K di chiavi memorizzate in un dizionario è molto più piccolo dell'universo U di tutte le possibili chiavi, una tabella hash richiede molto meno memoria di una tabella ad indirizzamento diretto. In particolare, la memoria richiesta può essere ridotta a $\Theta(|K|)$, e la ricerca di un elemento nella tabella hash richiede ancora solo tempo $O(1)$. (L'unico problema è che questo limite vale per il *tempo medio*, mentre per l'indirizzamento diretto vale per il *tempo nel caso peggiore*.)

Con l'indirizzamento diretto un elemento con chiave k è memorizzato nella posizione k. Con il metodo hash, questo elemento è memorizzato in $h(k)$; cioè viene usata una ***funzione hash*** h per calcolare la posizione a partire dalla chiave k: h definisce una corrispondenza tra l'universo U di chiavi e le posizioni di una ***tabella hash*** $T[0 \mathinner{..} m-1]$:

$$h: U \to \{0, 1, \ldots, m-1\}.$$

Se l'elemento con chiave k corrisponde alla posizione $h(k)$ si dice che $h(k)$ è il ***valore hash*** di k. La figura 12.2 illustra l'idea di base. Il punto essenziale della funzione hash è di ridurre l'intervallo degli indici dell'array che devono essere gestiti. Invece di $|U|$ valori, si devono gestire solo m valori. La memoria richiesta è ridotta in modo corrispondente.

Il difetto di quest'idea è che due chiavi possono corrispondere alla stessa posizione – fenomeno chiamato ***collisione***. Fortunatamente, vi sono tecniche efficaci per risolvere il conflitto creato dalle collisioni.

Figura 12.2 *Uso di una funzione hash h per porre in corrispondenza le chiavi e le posizioni nella tabella hash. Le chiavi k_2 e k_5 corrispondono alla stessa posizione, quindi collidono.*

Naturalmente la soluzione ideale dovrebbe essere di evitare del tutto le collisioni. Si potrebbe cercare di raggiungere questo obiettivo scegliendo una funzione hash h appropriata. Un'idea è quella di scegliere h in modo che appaia "casuale", evitando così collisioni o almeno minimizzando il loro numero. Il termine stesso "hash"[1], che richiama l'immagine di mischiare e tritare, rende bene l'idea di questo approccio. (Naturalmente una funzione hash h deve essere deterministica nel senso che un dato input k dovrebbe sempre produrre lo stesso output $h(k)$.) Poiché $|U| > m$, però, vi sono sicuramente due chiavi con lo stesso valore hash: evitare del tutto le collisioni è quindi impossibile. Perciò, seppure una funzione hash ben progettata, che sembri casuale, possa minimizzare il numero di collisioni, è ancora necessario un metodo per risolvere le collisioni che inevitabilmente si verificano.

Il resto del paragrafo presenta la tecnica di risoluzione delle collisioni più semplice, chiamata concatenazione. Il paragrafo 12.4 introduce un metodo alternativo per risolvere le collisioni chiamato indirizzamento aperto.

Risoluzione delle collisioni per concatenazione

Nel ***metodo di concatenazione***, si mettono tutti gli elementi che collidono nella stessa posizione in una lista concatenata, come mostrato nella figura 12.3. La posizione j contiene un puntatore alla testa della lista di tutti gli elementi memorizzati che corrispondono alla posizione j; se non vi sono elementi la posizione j contiene NIL.

Le operazioni del dizionario sono facili da realizzare con una tabella hash T quando le collisioni sono risolte per concatenazione.

CHAINED-HASH-INSERT(T, x)
  inserisci x in testa alla lista $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)
  ricerca un elemento con chiave k nella lista $T[h(k)]$

[1] N.d.T. Il termine *hash* è il nome che si dà al polpettone o più precisamente al piatto di carne avanzata del giorno prima e tritata insieme con verdure; data la difficoltà di tradurre questo termine esso è entrato così com'è nella terminologia informatica.

Figura 12.3 *Risoluzione delle collisioni per concatenazione. Ogni posizione $T[j]$ nella tabella hash contiene una lista concatenata di tutte le chiavi il cui valore hash è j. Per esempio, $h(k_1) = h(k_4)$ e $h(k_5) = h(k_2) = h(k_7)$.*

CHAINED-HASH-DELETE(T, k)
  cancella x dalla lista $T[h(key[x])]$

Il tempo di esecuzione per l'inserimento è $O(1)$ nel caso peggiore. Per la ricerca il tempo di esecuzione nel caso peggiore è proporzionale alla lunghezza della lista, come si analizzerà più precisamente fra poco. La cancellazione di un elemento x può richiedere tempo $O(1)$ se le liste sono bidirezionali. (Se le liste sono semplici, si deve prima trovare x nella lista $T[h(key[x])]$, così che il puntatore *next* del predecessore di x possa essere aggiornato rimuovendo x: in questo caso la cancellazione e la ricerca hanno essenzialmente lo stesso tempo di esecuzione.)

Analisi dell'organizzazione hash con concatenazione

Quali sono le prestazioni dell'organizzazione hash con concatenazione? In particolare, quanto tempo impiega la ricerca di un elemento con una data chiave?

Data una tabella hash T con m posizioni che memorizza n elementi, si definisce ***il fattore di carico*** α per T come n/m, cioè il *numero medio di elementi memorizzati in ogni lista concatenata*. L'analisi sarà espressa in termini di α, cioè si immagina che α rimanga fisso anche se n ed m tendono all'infinito (α può essere minore, maggiore o uguale a 1).

Il comportamento nel caso peggiore è sconfortante: tutte le n chiavi corrispondono alla stessa posizione creando una lista di lunghezza n. Così il tempo di ricerca nel caso peggiore è $\Theta(n)$ più il tempo per calcolare la funzione hash – tempo che non è migliore di quello che sarebbe necessario se si usasse una lista concatenata per tutti gli elementi. Chiaramente le tabelle hash non sono usate per le loro prestazioni nel caso peggiore.

Il comportamento medio dell'organizzazione hash dipende da quanto, in media, la funzione hash distribuisca bene l'insieme di chiavi da memorizzare sulle m posizioni. Il paragrafo 12.3 discute tale questione, ma per ora si assume che qualunque elemento corrisponda in modo equamente probabile a una delle m posizioni, indipendentemente dalla posizione degli altri elementi. Questa ipotesi si dice di ***uniformità semplice della funzione hash***.

Si assuma che il valore hash $h(k)$ possa essere calcolato in tempo $O(1)$, così che il tempo richiesto per cercare un elemento con chiave k dipenda in modo lineare dalla lunghezza della

lista $T[h(k)]$. Mettendo da parte il tempo $O(1)$ richiesto per calcolare la funzione hash e accedere alla posizione $h(k)$, si consideri il numero medio di elementi esaminati dall'algoritmo di ricerca, cioè il numero di elementi nella lista $T[h(k)]$ che sono controllati per verificare se le loro chiavi sono uguali a k. Considereremo due casi: nel primo, la ricerca è senza successo, cioè nessun elemento nella tabella ha chiave k, nel secondo, la ricerca con successo trova un elemento con chiave k.

Teorema 12.1

In una tabella hash in cui le collisioni sono risolte per concatenazione, nell'ipotesi di uniformità semplice della funzione hash, una ricerca senza successo richiede in media tempo $\Theta(1 + \alpha)$.

Dimostrazione. Nell'ipotesi di uniformità semplice della funzione hash, è equamente probabile che qualunque chiave k corrisponda a una qualunque delle m posizioni. Il tempo medio della ricerca senza successo di una chiave k è così il tempo medio richiesto per analizzare fino alla fine una delle m liste. La lunghezza media di una tale lista è il fattore di carico $\alpha = n/m$, per cui il numero medio di elementi esaminati in una ricerca senza successo è α, e il tempo totale richiesto (incluso il tempo per calcolare $h(k)$) è $\Theta(1 + \alpha)$. ■

Teorema 12.2

In una tabella hash in cui le collisioni sono risolte per concatenazione, nell'ipotesi di uniformità semplice della funzione hash, una ricerca con successo richiede in media tempo $\Theta(1 + \alpha)$.

Dimostrazione. Si assuma che la chiave che si sta cercando sia, in modo equamente probabile, una qualunque delle n chiavi memorizzate. Si assuma anche che la procedura CHAINED-HASH-INSERT inserisca un nuovo elemento in coda alla lista piuttosto che in testa. (Nell'Esercizio 12.2-3 si mostra che il tempo medio di ricerca con successo è lo stesso se i nuovi elementi sono inseriti in testa o in coda alla lista.) Il numero medio di elementi esaminati durante una ricerca con successo è 1 più il numero di elementi esaminati quando l'elemento cercato è stato inserito (poiché ogni nuovo elemento va alla fine della lista). Per trovare il numero medio di elementi esaminati, si prende la media, sugli n elementi della tabella, di 1 più la lunghezza media della lista in cui l'elemento i-esimo è aggiunto. La lunghezza media di tale lista è $(i-1)/m$, e così il numero medio di elementi esaminati in una ricerca con successo è

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)\\ &= 1+\left(\frac{1}{nm}\right)\left(\frac{(n-1)n}{2}\right)\\ &= 1+\frac{\alpha}{2}-\frac{1}{2m}\,.\end{aligned}$$

Per cui, il tempo totale richiesto per una ricerca con successo (incluso il tempo per calcolare la funzione hash) è $\Theta(2 + \alpha/2 - 1/2m) = \Theta(1 + \alpha)$. ■

Qual è il significato di questa analisi? Se il numero di posizioni nella tabella hash è almeno proporzionale al numero di elementi nella tabella, si ha $n = O(m)$ e, di conseguenza, $\alpha = n/m = O(m)/m = O(1)$. Quindi, la ricerca richiede in media tempo costante. Poiché l'inserzione richiede tempo $O(1)$ nel caso peggiore (si veda l'Esercizio 12.2-3) e la cancellazione, quando le liste sono bidirezionali, richiede tempo $O(1)$ nel caso peggiore, tutte le operazioni del dizionario possono essere eseguite in media con tempo $O(1)$.

Esercizi

12.2-1 Si supponga di usare una funzione hash casuale h per definire la corrispondenza di n chiavi distinte in un array T di lunghezza m. Qual è il numero medio di collisioni? Più precisamente, qual è la cardinalità media di $\{(x, y) : h(x) = h(y)\}$?

12.2-2 Si inseriscano le chiavi 5, 28, 19, 15, 20, 33, 12, 17, 10 in una tabella hash in cui le collisioni sono risolte per concatenazione. La tabella ha 9 posizioni e la funzione hash è $h(k) = k \bmod 9$.

12.2-3 Dedurre che il tempo medio per una ricerca con successo con il metodo della concatenazione è lo stesso se i nuovi elementi sono inseriti in testa o in coda alla lista. (*Suggerimento*: mostrare che il tempo medio di ricerca con successo è lo stesso per *qualunque* ordinamento di qualunque lista.)

12.2-4 Il professor Marley ipotizza che si possono ottenere dei miglioramenti sostanziali delle prestazioni se si modifica lo schema di concatenazione in modo che ogni lista sia mantenuta ordinata. La modifica del professore come modifica il tempo di esecuzione per la ricerca con successo, la ricerca senza successo, l'inserzione e la cancellazione?

12.2-5 Suggerire come allocare e deallocare memoria per gli elementi dentro la tabella hash stessa collegando in una lista libera tutte le posizioni inutilizzate. Si assuma che una posizione possa memorizzare una bandierina (o *flag*) e, inoltre, un elemento e un puntatore, oppure due puntatori. Tutte le operazioni del dizionario e della lista libera dovrebbero richiedere tempo medio $O(1)$. La lista libera deve essere bidirezionale o è sufficiente una lista semplice?

12.2-6 Mostrare che se $|U| > nm$, vi è un sottoinsieme di U di dimensione n che consiste di chiavi che corrispondono alla stessa posizione, così che l'organizzazione hash con concatenazione richiede nel caso peggiore un tempo di ricerca $\Theta(n)$.

12.3 Funzioni hash

In questo paragrafo saranno discusse alcune questioni che riguardano la definizione di buone funzioni hash e quindi si presentano tre schemi per la loro creazione: per divisione, per moltiplicazione e universale.

Requisiti di una buona funzione hash

Una buona funzione hash soddisfa (approssimativamente) l'ipotesi di uniformità semplice: ogni chiave corrisponde a ciascuna delle m posizioni in modo equamente probabile. Più formalmente, si assume che ogni chiave sia estratta in modo indipendente da U secondo la distribuzione di probabilità P; cioè $P(k)$ è la probabilità che k sia estratta. Allora l'ipotesi di uniformità semplice della funzione hash è che

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \quad \text{per } j = 0, 1, \ldots, m-1 . \tag{12.1}$$

Sfortunatamente, in generale non è possibile controllare questa condizione, poiché P di solito non è conosciuta.

Talvolta (ma raramente) si conosce la distribuzione P. Per esempio, si supponga di sapere che le chiavi sono numeri reali casuali distribuiti in modo indipendente e uniforme nell'intervallo $0 < k < 1$. In questo caso, si può mostrare che la funzione hash

$h(k) = \lfloor km \rfloor$

soddisfa l'equazione (12.1).

In pratica, possono essere usate tecniche euristiche per creare una funzione hash che si comporti bene con buona probabilità. Informazioni qualitative su P sono talvolta utili nel processo di definizione della funzione hash. Per esempio, si consideri la tabella dei simboli di un compilatore, in cui le chiavi sono stringhe qualsiasi di caratteri che rappresentano gli identificatori presenti in un programma. È un fatto comune che identificatori molto simili, come `pt` e `pts`, siano presenti in uno stesso programma. Una buona funzione hash dovrebbe minimizzare la possibilità che varianti dello stesso identificatore corrispondano alla stessa posizione.

Un approccio comune consiste nel derivare il valore hash in modo che sia indipendente da qualunque configurazione possa esistere nei dati. Per esempio, il "metodo di divisione" (discusso nel seguito) calcola il valore hash come il resto della divisione della chiave per uno specifico numero primo. A meno che il numero primo sia in qualche modo correlato con qualche particolarità nella distribuzione di probabilità, questo metodo dà buoni risultati.

Si noti infine che alcune applicazioni di funzioni hash potrebbero richiedere proprietà più forti di quelle fornite da funzioni hash uniformi. Per esempio, si potrebbe volere che chiavi "vicine" corrispondano a valori hash che siano "distanti". (La proprietà è desiderabile specialmente quando si usa la scansione lineare, definita nel paragrafo 12.4.)

Interpretazione delle chiavi come numeri naturali

Molte funzioni hash assumono che l'universo delle chiavi sia l'insieme $\mathbb{N} = \{0, 1, 2, \ldots\}$ dei numeri naturali. Per cui, se le chiavi non sono numeri naturali, bisogna trovare un modo per interpretarle come tali. Per esempio, una chiave che è una stringa di caratteri può essere interpretata come un intero espresso in rappresentazione posizionale in una base adatta. Così l'identificatore `pt` potrebbe essere interpretato come la coppia di interi in rappresentazione decimale (112, 116), poiché `p` = 112 e `t` = 116 nella codifica ASCII; quindi, espresso come un intero in base 128, `pt` diventa $(112 \cdot 128) + 116 = 14452$. Di solito in qualunque applicazione è facile escogitare qualche semplice metodo tipo questo per interpretare ogni chiave come un numero naturale (anche grande). Nel seguito si assume che le chiavi siano numeri naturali.

12.3.1 Il metodo di divisione

Nel *metodo di divisione* per definire funzioni hash, si fa corrispondere una chiave k a una delle m posizioni, prendendo il resto della divisione di k per m. Cioè la funzione hash è

$h(k) = k \bmod m$.

Per esempio, se la tabella hash ha dimensione $m = 12$ e la chiave è $k = 100$, allora $h(k) = 4$. Poiché è richiesta solo un'operazione di divisione, il calcolo della funzione hash è abbastanza veloce.

Quando si usa il metodo di divisione, di solito si evitano certi valori di m. Per esempio m non dovrebbe essere potenza di 2, poiché se $m = 2^p$, allora $h(k)$ è dato dai p bit meno significativi di k. A meno che si sappia a priori che la distribuzione di probabilità sulle chiavi prevede come equamente probabili tutte le configurazioni dei p bit meno significativi, è meglio scegliere la funzione hash in modo che dipenda da tutti i bit della chiave. Le potenze di 10 dovrebbero essere evitate se l'applicazione tratta numeri decimali come chiavi, poiché altrimenti la funzione hash non dipenderebbe da tutte le cifre decimali di k. Infine si può mostrare che quando $m = 2^p - 1$ e k è una stringa di caratteri interpretata in base 2^p, due stringhe identiche eccetto che per uno scambio tra due caratteri adiacenti corrisponderanno allo stesso valore.

Buoni valori di m sono valori primi non troppo vicini a potenze esatte di 2. Per esempio, si supponga di voler allocare una tabella hash, con concatenazione, per contenere $n = 2000$ stringhe di caratteri, dove un carattere richiede 8 bit. Esaminare in media 3 elementi in una ricerca senza successo può essere soddisfacente, così si alloca una tabella hash di dimensione $m = 701$. Questo numero è stato scelto perché è un numero primo vicino a 2000/3 ma non vicino ad alcuna potenza di 2. Trattando ogni chiave k come un intero, la funzione hash sarebbe

$h(k) = k \bmod 701$.

Per precauzione, si potrebbe controllare quanto questa funzione hash distribuisce regolarmente l'insieme di chiavi tra le posizioni, provando con chiavi scelte su dati "reali".

12.3.2 Il metodo di moltiplicazione

Il *metodo di moltiplicazione* per definire funzioni hash opera in due passi. Prima si moltiplica la chiave k per una costante A nell'intervallo $0 < A < 1$ e si estrae la parte frazionaria di kA. Quindi si moltiplica questo valore per m e si prende la parte intera del risultato. In breve, la funzione hash è

$h(k) = \lfloor m(kA \bmod 1) \rfloor$,

dove "$kA \bmod 1$" rappresenta la parte frazionaria di kA, cioè $kA - \lfloor kA \rfloor$.

Un vantaggio del metodo di moltiplicazione è che il valore di m non è critico. Tipicamente si sceglie una potenza di 2, cioè $m = 2^p$ per qualche intero p, poiché così si può facilmente realizzare la funzione su molti calcolatori nel modo seguente. Si supponga che la dimensione della parola della macchina sia di w bit e che k entri in una sola parola. Con riferimento alla figura 12.4, prima si moltiplica k per l'intero di w bit $\lfloor A \cdot 2^w \rfloor$. Il risultato è un valore su $2w$ bit pari a $r_1 2^w + r_0$, dove r_1 è la parola più significativa del prodotto e r_0 è quella meno significativa. Il valore hash desiderato di p bit consiste dei p bit più significativi di r_0.

Figura 12.4 *Il metodo di moltiplicazione per definire una funzione hash. La rappresentazione con w bit della chiave k è moltiplicata per il valore su w bit pari a* $\lfloor A \cdot 2^w \rfloor$, *dove* $0 < A < 1$ *è una costante opportuna. I p bit più significativi della metà meno significativa del prodotto formano il valore hash desiderato* $h(k)$.

Sebbene questo metodo funzioni per qualunque valore della costante A, con qualche valore funziona meglio che con altri. La scelta ottima dipende dalle caratteristiche dei dati su cui vengono prese le chiavi. Knuth [123] discute la scelta di A in dettaglio e suggerisce che

$$A = (\sqrt{5} - 1)/2 = 0.6180339887\ldots \tag{12.2}$$

è probabile che funzioni ragionevolmente bene.

Per esempio, se si ha $k = 123456$, $m = 10000$ e A come descritto nell'equazione (12.2), allora

$$\begin{aligned} h(k) &= \lfloor 10000 \cdot (123456 \cdot 0.61803\ldots \bmod 1) \rfloor \\ &= \lfloor 10000 \cdot (76300.0041151\ldots \bmod 1) \rfloor \\ &= \lfloor 10000 \cdot 0.0041151\ldots \rfloor \\ &= \lfloor 41.151\ldots \rfloor \\ &= 41 . \end{aligned}$$

12.3.3 Funzione hash universale

Se un avversario dispettoso scegliesse le chiavi su cui calcolare la funzione hash, allora potrebbe scegliere n chiavi che corrispondano tutte alla stessa posizione, provocando un tempo medio di ricerca $\Theta(n)$. Qualunque funzione hash è sottoposta a questo tipo di comportamento nel caso peggiore; il solo modo efficace di migliorare la situazione è di scegliere una funzione hash in *modo casuale* così che sia *indipendente* dalle chiavi che vanno effettivamente memorizzate. Quest'approccio, chiamato ***hashing universale***, mantiene una buona prestazione nel caso medio, a prescindere dalle chiavi scelte dall'avversario.

L'idea di base di quest'approccio è di selezionare, in modo casuale, la funzione hash al tempo di esecuzione da una classe di funzioni attentamente definite. Come nel caso del quicksort, la scelta casuale garantisce che nessun input particolare provochi sempre il comportamento del caso peggiore. A causa della scelta casuale, l'algoritmo si può comportare diversamente ad ogni esecuzione, anche per lo stesso input. Quest'approccio garantisce una buona prestazione nel caso medio, a prescindere dalle chiavi fornite in input. Ritornando all'esempio della tabella dei simboli di un compilatore, si trova che la scelta degli identificatori da parte del programmatore in tal caso non può causare costantemente una cattiva prestazione della funzione hash. Una cattiva prestazione si ha solo se il compilatore sceglie una funzione

hash casuale che mette male in corrispondenza gli identificatori con le posizioni, ma la probabilità che ciò accada è piccola ed è la stessa per qualunque insieme di identificatori della stessa dimensione.

Sia $\mathcal{H}$ un insieme finito di funzioni hash che vanno da un dato universo U di chiavi all'intervallo $\{0, 1, \ldots, m-1\}$. Tale insieme è detto ***universale*** se per ogni coppia di chiavi distinte $x, y \in U$, il numero di funzioni hash $h \in \mathcal{H}$ per cui $h(x) = h(y)$ è precisamente $|\mathcal{H}|/m$. In altre parole, con una funzione hash scelta in modo casuale da $\mathcal{H}$, la probabilità di una collisione tra x e y per $x \neq y$ è esattamente $1/m$, che è esattamente la probabilità di una collisione se $h(x)$ e $h(y)$ sono scelte casualmente nell'insieme $\{0, 1, \ldots, m-1\}$.

Il seguente teorema mostra che una classe universale di funzioni hash fornisce un comportamento buono nel caso medio.

Teorema 12.3

Se h è scelta da un insieme universale di funzioni hash ed è usata per la corrispondenza di n chiavi su una tabella di dimensione m, dove $n \leq m$, il numero medio di collisioni che coinvolge una particolare chiave x è minore di 1.

Dimostrazione. Per ogni coppia y, z di chiavi distinte, sia c_{yz} una variabile casuale che è 1 se $h(y) = h(z)$ (cioè se y e z collidono usando h) e 0 altrimenti. Poiché, per definizione, una singola coppia di chiavi collide con probabilità $1/m$, si ha

$$E[c_{yz}] = 1/m.$$

Sia C_x il numero totale di collisioni che coinvolgono la chiave x in una tabella hash T di dimensione m contenente n chiavi. L'equazione (6.24) fornisce

$$\begin{aligned} E[C_x] &= \sum_{\substack{y \in T \\ y \neq x}} E[c_{xy}] \\ &= \frac{n-1}{m} . \end{aligned}$$

Poiché $n \leq m$, si ha $E[C_x] < 1$. ■

Ma quanto è semplice definire una classe universale di funzioni hash? È abbastanza facile, come un po' di teoria ci aiuterà a provare. Si scelga come dimensione m della tabella un numero primo (come nel metodo di divisione). Si decompone una chiave x in $r+1$ byte (cioè caratteri o sottostringhe binarie di ampiezza fissa), così che $x = \langle x_0, x_1, \ldots, x_r \rangle$; la sola richiesta è che il massimo valore di un byte sia minore di m. Si denoti con $a = \langle a_0, a_1, \ldots, a_r \rangle$ una sequenza di $r+1$ elementi scelti a caso dall'insieme $\{0, 1, \ldots, m-1\}$. Si definisce una corrispondente funzione hash $h_a \in \mathcal{H}$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m . \tag{12.3}$$

Con questa definizione,

$$\mathcal{H} = \bigcup_a \{h_a\} \tag{12.4}$$

ha m^{r+1} elementi.

Teorema 12.4

La classe $\mathcal{H}$ definita dalle equazioni (12.3) e (12.4) è una classe universale di funzioni hash.

Dimostrazione. Si consideri qualunque coppia di chiavi distinte x, y. Si assuma che $x_0 \neq y_0$. (Si procede in modo analogo se la differenza è in uno qualunque degli altri byte.) Per qualunque valore fissato di $a_1, a_2, \ldots, a_r$ vi è esattamente un valore di a_0 che soddisfa l'equazione $h(x) = h(y)$; questo a_0 è la soluzione di

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^{r} a_i(x_i - y_i) \pmod{m} .$$

Per verificare questa proprietà, si noti che, poiché m è primo, la quantità non nulla $x_0 - y_0$ ha un inverso moltiplicativo modulo m, e così vi è un'unica soluzione per a_0 modulo m (si veda il paragrafo 33.4). Quindi, ogni coppia di chiavi x e y collide esattamente per m^r valori di a, poiché essi collidono esattamente una volta per ogni possibile valore di $\langle a_1, a_2, \ldots, a_r \rangle$ (cioè per l'unico valore di a_0, mostrato sopra). Poiché vi sono m^{r+1} possibili valori per la sequenza a, le chiavi x e y collidono esattamente con probabilità $m^r/m^{r+1} = 1/m$. Quindi $\mathcal{H}$ è universale. ■

Esercizi

12.3-1 Si supponga di voler eseguire una ricerca su una lista concatenata di lunghezza n, dove ogni elemento contiene una chiave k insieme con un valore hash $h(k)$. Ogni chiave è una lunga stringa di caratteri. Come si potrebbe trarre vantaggio dalla presenza dei valori hash nella ricerca di una data chiave nella lista?

12.3-2 Si supponga di avere una corrispondenza tra una stringa di r caratteri e m posizioni, dove la stringa è trattata come un numero in base 128, e di usare il metodo di divisione. Il numero m è facilmente rappresentato come una parola di macchina di 32 bit, ma la stringa di r caratteri, trattata come un numero in base 128, richiede molte parole. Come si può applicare il metodo di divisione per calcolare il valore hash della stringa di caratteri senza usare più di un numero costante di parole di memoria al di fuori della stringa stessa?

12.3-3 Si consideri una versione del metodo di divisione in cui $h(k) = k \bmod m$, dove $m = 2^{p-1}$ e k è una stringa di caratteri rappresentata in base 2^p. Mostrare che se la stringa x può essere derivata da una permutazione dei caratteri della stringa y allora x e y corrispondono allo stesso valore hash. Dare un esempio di applicazione in cui questa proprietà crea delle difficoltà in una funzione hash.

12.3-4 Si consideri una tabella hash di dimensione $m = 1000$ e una funzione hash $h(k) = \lfloor m(kA \bmod 1) \rfloor$ per $A = (\sqrt{5} - 1)/2$. Calcolare le posizioni corrispondenti alle chiavi 61, 62, 63, 64 e 65.

12.3-5 Mostrare che se si obbliga ogni componente a_i di a nell'equazione (12.3) a essere diversa da 0, allora l'insieme $\mathcal{H} = \{h_a\}$ definito nell'equazione (12.4) non è universale. (*Suggerimento*: si considerino le chiavi $x = 0$ e $y = 1$.)

12.4 Indirizzamento aperto

Nell'*indirizzamento aperto* tutti gli elementi sono memorizzati nella tabella hash stessa. Cioè ogni elemento della tabella contiene o un elemento dell'insieme dinamico o NIL. Quando si ricerca un elemento, si esaminano sistematicamente le posizioni della tabella finché o si trova l'elemento desiderato o è chiaro che non è nella tabella. Non vi sono liste né elementi memorizzati fuori della tabella, come succede per la concatenazione. Quindi nell'indirizzamento aperto la tabella hash può riempirsi finché nessun altro elemento può essere inserito; il fattore di carico non può mai superare 1.

Naturalmente, si potrebbero memorizzare liste per concatenazione all'interno della tabella hash, nelle posizioni altrimenti inutilizzate della tabella (si veda l'Esercizio 12.2-5), ma il vantaggio dell'indirizzamento aperto è di evitare del tutto i puntatori. Invece di seguire i puntatori, si *calcola* la sequenza di posizioni da esaminare. La memoria extra liberata dai puntatori non memorizzati fornisce più posizioni per la tabella a parità di occupazione di memoria, consentendo potenzialmente poche collisioni e un recupero più veloce.

Per eseguire l'inserzione usando l'indirizzamento aperto, si esaminano in successione le posizioni della tabella, cioè si esegue una *scansione*, finché si trova una posizione vuota in cui inserire la chiave. Invece di essere fissata dalla sequenza $0, 1, \ldots, m-1$ (che richiede tempo di ricerca $\Theta(n)$), la sequenza di posizioni esaminate *dipende dalla chiave che deve essere inserita*. Per determinare la posizione da esaminare, si estende la funzione hash perché includa *il numero di posizioni già esaminate* (che parte da 0) come secondo input. Così la funzione hash diventa

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\} .$$

Con l'indirizzamento aperto si richiede che per ogni chiave k, la *sequenza di scansione*

$$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

sia una permutazione di $\langle 0, 1, \ldots, m-1 \rangle$, così che ogni elemento della tabella hash sia eventualmente considerato come una posizione per la nuova chiave fino a riempire tutta la tabella. Nel seguente pseudocodice, si assume che gli elementi nella tabella T siano chiavi senza informazioni satellite; la chiave k è proprio l'elemento stesso che la contiene. Ogni posizione contiene o una chiave o NIL (se la posizione è vuota).

```
HASH-INSERT(T, k)
  i ← 0
  repeat j ← h(k, i)
          if T[j] = NIL
             then T[j] ← k
                  return j
             else i ← i + 1
     until i = m
  error "overflow sulla tabella hash"
```

L'algoritmo di ricerca della chiave k scandisce la stessa sequenza di posizioni esaminata dall'algoritmo di inserzione quando k è stata inserita. Quindi, la ricerca può terminare (senza successo) quando trova una posizione vuota, poiché k sarebbe stata inserita proprio lì e non in una successiva posizione della sequenza di scansione. (Si noti che questa asserzione assume che le chiavi non vengano cancellate dalla tabella hash.) La procedura HASH-SEARCH prende come input una tabella T e una chiave k, restituendo j se la posizione j contiene la chiave k, o NIL se la chiave non è presente nella tabella T.

```
HASH-SERCH(T, k)
  i ← 0
  repeat j ← h(k, i)
          if T[j] = k
             then return j
          i ← i + 1
     until T[j] = NIL o i = m
  return NIL
```

L'operazione di cancellazione da una tabella hash ad indirizzamento aperto è difficile. Quando si cancella una chiave da una posizione i, non si può semplicemente marcare quella posizione come vuota memorizzandovi NIL. Così facendo potrebbe essere impossibile recuperare qualunque chiave k per la quale, durante l'inserzione, la posizione i era stata esaminata ed era stata trovata occupata. Una soluzione è di marcare la posizione memorizzandovi il valore speciale DELETED invece di NIL. Si dovrebbe allora modificare la procedura HASH-INSERT perché tratti una tale posizione come se fosse vuota in modo da inserirvi una nuova chiave. Così facendo, però, i tempi di ricerca non dipendono più soltanto dal fattore di carico α. Per questa ragione, quando le chiavi devono essere cancellate, di solito si sceglie la tecnica di risoluzione delle collisioni per concatenazione.

Nell'analisi si fa l'ipotesi di ***uniformità della funzione hash***: ogni chiave considerata ha, in modo equamente probabile, una qualunque delle $m!$ permutazioni di $\{0, 1, \ldots, m-1\}$ come sequenza di scansione. L'uniformità della funzione hash generalizza la nozione di uniformità semplice, definita prima, alla situazione in cui la funzione hash produce non un singolo valore ma un'intera sequenza di scansione. Però, in verità, una funzione hash uniforme è difficile da realizzare e in pratica vengono usate delle opportune approssimazioni (come l'hashing doppio, definito nel seguito).

Tre sono le tecniche usate di solito per calcolare la sequenza di scansione richiesta per l'indirizzamento aperto: scansione lineare, scansione quadratica e hashing doppio. Tutte queste tecniche garantiscono che $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ sia una permutazione di $\langle 0, 1, \ldots, m-1\rangle$ per ogni chiave k. Nessuna di queste tecniche però rispetta l'ipotesi di uniformità poiché nessuna di esse è in grado di generare più di m^2 sequenze di scansione diverse (invece delle $m!$ richieste nell'ipotesi di uniformità). L'hashing doppio ha il maggior numero di sequenze di scansione e, come ci si potrebbe aspettare, dà in pratica i risultati migliori.

Scansione lineare

Data una funzione hash $h' : U \rightarrow \{0, 1, \ldots, m-1\}$, il metodo di ***scansione lineare*** usa la funzione hash

$$h(k, i) = (h'(k) + i) \bmod m$$

per $i = 0, 1, \ldots, m-1$. Data la chiave k, la prima posizione esaminata è $T[h'(k)]$. Quindi si scandisce la posizione $T[h'(k) + 1]$, e così via fino a scandire la posizione $T[m-1]$. Quindi si torna in modo circolare a scandire le posizioni $T[0], T[1], \ldots$, finché si raggiunge $T[h'(k) - 1]$. Poiché la posizione iniziale della scansione determina l'intera sequenza di scansione, con la scansione lineare sono usate solo m diverse sequenze di scansione.

La scansione lineare è facile da realizzare, ma presenta un problema conosciuto come fenomeno di ***agglomerazione primaria***. Le posizioni occupate si accumulano in lunghi tratti, aumentando il tempo medio di ricerca. Per esempio, se vi sono $n = m/2$ chiavi nella tabella, dove ogni posizione con indice pari è occupata e ogni posizione con indice dispari è vuota, allora la ricerca senza successo richiede in media 1,5 accessi. Però se le posizioni occupate sono le prime $n = m/2$ locazioni, il numero medio di accessi cresce a circa $n/4 = m/8$. Di solito gli agglomerati si verificano perché se una posizione vuota è preceduta da i posizioni piene, allora la probabilità che la posizione vuota sia la prossima a essere riempita è $(i + 1)/m$, in confronto a una probabilità di $1/m$ se la precedente posizione era vuota. Per cui, tratti di posizioni occupate tendono a diventare ancora più lunghi e la scansione lineare non è una buona approssimazione di una funzione hash uniforme.

Scansione quadratica

La ***scansione quadratica*** usa una funzione hash della forma

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \tag{12.5}$$

dove (come nella scansione lineare) h' è una funzione hash ausiliaria, c_1 e $c_2 \neq 0$ sono costanti ausiliarie e $i = 0, 1, \ldots, m-1$. La prima posizione esaminata è $T[h'(k)]$; le posizioni esaminate successivamente sono distanziate di una quantità che dipende in modo quadratico dal numero i di accessi. Questo metodo funziona molto meglio della scansione lineare, ma i valori di c_1, c_2 e m sono vincolati per poter fare uso dell'intera tabella hash.

Il Problema 12-4 mostra un modo per selezionare questi parametri. Inoltre, se due chiavi hanno la stessa posizione iniziale, allora le loro sequenze di scansione sono le stesse, poiché $h(k_1, 0) = h(k_2, 0)$ implica $h(k_1, i) = h(k_2, i)$. Questo porta a una forma più sottile di agglomerazione, chiamata ***agglomerazione secondaria***. Come per la scansione lineare, l'accesso iniziale determina l'intera sequenza, quindi sono usate solo m distinte sequenze di scansione.

Hashing doppio

L'hashing doppio è uno dei migliori metodi disponibili per l'indirizzamento aperto perché le permutazioni prodotte hanno molte delle caratteristiche delle permutazioni scelte casualmente. L'***hashing doppio*** usa una funzione hash della forma

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m,$$

Figura 12.5 *Inserzione con hashing doppio. In questo caso si ha una tabella hash di dimensione* 13 *con* $h_1(k) = k \bmod 13$ *e* $h_2(k) = 1 + (k \bmod 11)$. *Poiché* $14 \equiv 1 \bmod 13$ *e* $14 \equiv 3 \bmod 11$, *la chiave* 14 *sarà inserita nella posizione* 9 *che è vuota, dopo che le posizioni* 1 *e* 5 *siano state esaminate e trovate già occupate.*

dove h_1 e h_2 sono funzioni hash ausiliarie. La prima posizione esaminata è $T[h_1(k)]$; ogni posizione esaminata successivamente è distanziata dalla precedente posizione di una quantità $h_2(k)$, modulo m. Quindi, diversamente dal caso della scansione lineare o quadratica, in questo caso la sequenza di scansione dipende dalla chiave k in due modi, poiché la posizione iniziale, la distanza, o entrambe, possono variare a seconda della chiave. La figura 12.5 fornisce un esempio di inserzione con l'hashing doppio.

Il valore $h_2(k)$ deve essere primo rispetto alla dimensione m della tabella hash per qualunque chiave da ricercare. Altrimenti se m e $h_2(k)$ avessero un massimo comune divisore $d > 1$ per qualche chiave k, allora una ricerca di tale chiave andrebbe ad esaminare solo una frazione $(1/d)$ della tabella hash. (Si veda il Capitolo 33.) Un modo comodo di assicurare questa condizione è di scegliere m come potenza di 2 e definire h_2 in modo che produca sempre un numero dispari. Un altro modo è scegliere m primo e h_2 in modo che restituisca sempre un intero positivo minore di m. Per esempio si può scegliere m primo e porre:

$$h_1(k) = k \bmod m,$$
$$h_2(k) = 1 + (k \bmod m'),$$

dove m' è scelto in modo che sia di poco minore di m (per esempio $m-1$ o $m-2$). Per esempio, se $k = 123456$, $m = 701$ e $m' = 700$, si ha $h_1(k) = 80$ e $h_2(k) = 257$, così il primo accesso è alla posizione 80 e quindi viene esaminata una posizione ogni 257 posizioni (modulo m) finché si trova la chiave o sono state esaminate tutte le posizioni.

L'hashing doppio rappresenta un miglioramento della scansione lineare o quadratica per il fatto che vengono utilizzate $\Theta(m^2)$ sequenze di scansione, piuttosto che $\Theta(m)$, poiché ogni possibile coppia $(h_1(k), h_2(k))$ implica una diversa sequenza di scansione e, a seconda della chiave, la posizione iniziale $h_1(k)$ e la distanza $h_2(k)$ possono variare in modo indipendente. Ne risulta che il comportamento dell'hashing doppio sembra essere molto vicino a quello dello schema "ideale" della funzione hash uniforme.

Analisi dell'organizzazione hash con indirizzamento aperto

L'analisi dell'indirizzamento aperto, come quella della concatenazione, viene espressa in termini del fattore di carico α della tabella hash, per n ed m che tendono all'infinito. Si ricorda che se n elementi sono memorizzati in una tabella di m posizioni, il numero medio di elementi per ogni posizione è $\alpha = n/m$. Naturalmente, con l'indirizzamento aperto, si ha al più un elemento per posizione, e quindi $n \le m$, che implica $\alpha \le 1$.

Si assuma che venga usata una funzione hash uniforme. In questo schema ideale, la sequenza di scansione $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ per ogni chiave k è, in modo equamente probabile, una qualunque permutazione di $\langle 0, 1, \ldots, m-1\rangle$. Cioè ogni possibile sequenza di scansione è usata in modo equiprobabile per un'inserzione o una ricerca. Naturalmente una data chiave ha associata una sequenza di scansione stabilita, il che significa che considerando la distribuzione di probabilità sullo spazio di chiavi e l'operazione della funzione hash sulle chiavi, ogni possibile sequenza di scansione è equamente probabile.

Si analizza ora il numero medio di accessi in un'organizzazione hash a indirizzamento aperto nell'ipotesi di uniformità della funzione hash, cominciando dall'analisi del numero di accessi fatti in una ricerca senza successo.

Teorema 12.5

Data una tabella hash a indirizzamento aperto con fattore di carico $\alpha = n/m < 1$, il numero medio di accessi di una *ricerca senza successo* è al più $1/(1-\alpha)$, assumendo l'uniformità della funzione hash.

Dimostrazione. In una ricerca senza successo ogni accesso tranne l'ultimo è a una posizione occupata che non contiene la chiave desiderata e l'ultima posizione considerata è vuota. Si definisca

$p_i = \Pr\{\text{esattamente } i \text{ accessi a posizioni occupate}\}$

per $i = 0, 1, 2, \ldots$. Per $i > n$ si ha $p_i = 0$, poiché si possono trovare al più n posizioni già occupate, per cui il numero medio di accessi è

$$1 + \sum_{i=0}^{\infty} i\, p_i \,. \tag{12.6}$$

Per valutare l'equazione (12.6), si definisce

$q_i = \Pr\{\text{almeno } i \text{ accessi a posizioni occupate}\}$

per $i = 0, 1, 2, \ldots$. Si può quindi usare l'identità (6.28):

$$\sum_{i=0}^{\infty} i\, p_i = \sum_{i=1}^{\infty} q_i \,.$$

Qual è il valore di q_i per $i \ge 1$? La probabilità che la prima posizione cui si accede sia occupata è n/m; per cui,

$$q_1 = \frac{n}{m} \,.$$

Con una funzione hash uniforme, un secondo accesso, se necessario, è sulle rimanenti $m-1$ posizioni ancora non esaminate, $n-1$ delle quali sono occupate. Si fa un secondo accesso solo se il primo è ad una posizione occupata; per cui,

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right) .$$

In generale, l'i-esimo accesso è fatto solo se i primi $i-1$ accessi sono a posizioni occupate e la posizione esaminata è in modo equamente probabile una qualunque delle restanti $m-i+1$ posizioni, $n-i+1$ delle quali sono occupate. Per cui,

$$\begin{aligned} q_i &= \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+1}{m-i+1}\right) \\ &\le \left(\frac{n}{m}\right)^i \\ &= \alpha^i \end{aligned}$$

per $i = 1, 2, \ldots, n$, poiché $(n-j)/(m-j) \le n/m$ se $n \le m$ e $j \ge 0$. Dopo n accessi, tutte le n posizioni occupate sono state esaminate e non saranno più scandite, quindi $q_i = 0$ per $i > n$.

Si può ora valutare l'equazione (12.6). Data l'ipotesi che $\alpha < 1$, il numero medio di accessi di una ricerca senza successo è

$$\begin{aligned} 1 + \sum_{i=0}^{\infty} i\,p_i &= 1 + \sum_{i=1}^{\infty} q_i \\ &\le 1 + \alpha + \alpha^2 + \alpha^3 + \cdots \\ &= \frac{1}{1-\alpha} . \end{aligned} \tag{12.7}$$

L'equazione (12.7) ha un'interpretazione intuitiva: un accesso è sempre fatto, con probabilità α ne è necessario un secondo, con probabilità approssimativamente α^2 ne è necessario un terzo e così via. ■

Se α è una costante, il Teorema 12.5 pronostica che una ricerca senza successo richieda tempo $O(1)$. Per esempio, se la tabella hash è mezza piena, il numero medio di accessi in una ricerca senza successo è al più $1/(1-0.5) = 2$. Se è piena al 90 per cento, il numero medio di accessi è al più $1/(1-0.9) = 10$.

Il Teorema 12.5 fornisce le prestazioni della procedura HASH-INSERT in modo immediato.

Corollario 12.6

L'inserzione di un elemento in una tabella hash ad indirizzamento aperto con fattore di carico α richiede al più $1/(1-\alpha)$ accessi in media, nell'ipotesi di usare una funzione hash uniforme.

Dimostrazione. Un elemento viene inserito solo se vi è spazio nella tabella, perciò $\alpha < 1$. L'inserzione di una chiave richiede una ricerca senza successo seguita dalla memorizzazione della chiave nella prima posizione trovata vuota. Quindi il numero medio di accessi è al più $1/(1-\alpha)$. ■

Il calcolo del numero medio di accessi in una ricerca con successo è un po' più laborioso.

Teorema 12.7

Data una tabella hash ad indirizzamento aperto con fattore di carico $\alpha < 1$, il numero medio di accessi in una *ricerca con successo* è al più

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha} ,$$

nell'ipotesi di usare una funzione hash uniforme e assumendo che ogni chiave sia ricercata nella tabella in modo equiprobabile.

Dimostrazione. La ricerca di una chiave k segue la stessa sequenza di scansione seguita per l'inserzione dell'elemento con chiave k. Dal Corollario 12.6, se k è la $(i+1)$-esima chiave inserita nella tabella hash, il numero medio di accessi fatti in una ricerca di k è al più $1/(1-i/m) = m/(m-i)$. Eseguendo la media su tutte le n chiavi nella tabella hash si ha il numero medio di accessi in una ricerca con successo:

$$\begin{aligned} \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n}) , \end{aligned}$$

dove $H_i = \sum_{j=1}^{i} 1/j$ è l'i-esimo numero armonico (come definito nell'equazione (3.5)). Usando i limiti $\ln i \le H_i \le \ln i + 1$ dalle equazioni (3.11) e (3.12), si ottiene

$$\begin{aligned} \frac{1}{\alpha}(H_m - H_{m-n}) &= \frac{1}{\alpha}\sum_{k=m-n+1}^{m} 1/k \\ &\le \frac{1}{\alpha}\int_{m-n}^{m}(1/x)\,dx \\ &= \frac{1}{\alpha}\ln(m/(m-n)) \\ &= \frac{1}{\alpha}\ln(1/(1-\alpha)) \end{aligned}$$

come limite sul numero medio di accessi in una ricerca con successo. ■

Se la tabella hash è piena a metà, il numero medio di accessi in una ricerca con successo è meno di 1.387. Se la tabella hash è piena al 90 per cento, il numero medio di accessi è meno di 2.559.

Esercizi

12.4-1 Si consideri l'inserzione delle chiavi 10, 22, 31, 4, 15, 28, 17, 88, 59 in una tabella hash di lunghezza $m = 11$ usando l'indirizzamento aperto con funzione hash primaria $h'(k) = k \bmod m$. Si illustri il risultato dell'inserimento di queste chiavi usando la scansione lineare, usando la scansione quadratica con $c_1 = 1$ e $c_2 = 3$ e usando l'hashing doppio con $h_2(k) = 1 + (k \bmod (m-1))$.

12.4-2 Scrivere in pseudocodice la procedura HASH-DELETE come delineata nel testo e modificare HASH-INSERT e HASH-SEARCH perché prevedano il valore speciale DELETED.

* *12.4-3* Si supponga di risolvere le collisioni con un hashing doppio; cioè si usi la funzione hash $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Mostrare che la sequenza di scansione $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ è una permutazione della sequenza di posizioni $\langle 0, 1, \ldots, m-1\rangle$ se e solo se $h_2(k)$ è primo rispetto ad m. (*Suggerimento*: si veda il Capitolo 33.)

12.4-4 Si supponga di usare una funzione hash uniforme e una tabella hash ad indirizzamento aperto con fattore di carico $\alpha = 1/2$. Qual è il numero medio di accessi in una ricerca senza successo? Qual è il numero medio di accessi in una ricerca con successo? Ripetere i calcoli per i fattori di carico 3/4 e 7/8.

* *12.4-5* Si supponga di inserire n chiavi in una tabella hash di dimensione m usando l'indirizzamento aperto e una funzione hash uniforme. Sia $p(n, m)$ la probabilità che non si verifichi alcuna collisione. Mostrare che $p(n, m) \le e^{-n(n-1)/2m}$. (*Suggerimento*: si veda l'equazione (2.7).) Dedurre che quando n eccede $\sqrt{m}$ la probabilità di evitare le collisioni va rapidamente a zero.

* *12.4-6* Si consideri una tabella hash ad indirizzamento aperto con fattore di carico α. Trovare il valore non nullo di α per cui il numero medio di accessi di una ricerca senza successo sia uguale a due volte il numero medio di accessi di una ricerca con successo. Usare la stima $(1/\alpha)\ln(1/(1-\alpha))$ per il numero medio di accessi di una ricerca con successo.

Problemi

12-1 Limite alla scansione più lunga in un'organizzazione hash

Una tabella hash di dimensione m è usata per memorizzare n elementi, con $n \le m/2$. L'indirizzamento aperto è usato per la risoluzione delle collisioni.

a. Assumendo l'uniformità della funzione hash, mostrare che per $i = 1, 2, \ldots, n$, la probabilità che l'i-esima inserzione richieda strettamente più di k accessi è al più 2^{-k}.

b. Mostrare che per $i = 1, 2, \ldots, n$, la probabilità che l'i-esima inserzione richieda più di $2 \lg n$ accessi è al più $1/n^2$.

La variabile casuale X_i denoti il numero di accessi richiesti dall'i-esima inserzione. Nella parte (b) si è già mostrato che $\Pr\{X_i > 2 \lg n\} \le 1/n^2$. La variabile casuale $X = \max_{1 \le i \le n} X_i$ denoti il massimo numero di accessi richiesti da una qualunque delle n inserzioni.

c. Mostrare che $\Pr\{X \ge 2 \lg n\} \le 1/n$.

d. Mostrare che la lunghezza media della sequenza di scansione più lunga è $E[X] = O(\lg n)$.

12-2 Ricerca in un insieme statico

Finora si è fatta l'ipotesi di lavorare con un insieme dinamico di n elementi in cui le chiavi siano numeri. Si consideri ora l'insieme statico che non richiede le operazioni INSERT e DELETE ma la sola operazione SEARCH. Sia disponibile una quantità arbitraria di tempo per eseguire una preelaborazione sugli n elementi in modo che l'operazione SEARCH sia eseguita rapidamente.

a. Mostrare che SEARCH può essere realizzata in tempo $O(\lg n)$ nel caso peggiore, senza usare memoria extra oltre quella necessaria a memorizzare gli elementi dell'insieme.

b. Si supponga di realizzare l'insieme con una tabella hash a indirizzamento aperto di m posizioni, con funzione hash uniforme. Qual è la minima quantità $m - n$ di memoria extra richiesta perché il tempo medio di una SEARCH senza successo sia almeno altrettanto buono quanto il limite della parte (a)? La risposta dovrebbe essere un limite asintotico su $m - n$ in termini di n.

12-3 Limite alla dimensione della tabella nella concatenazione

Si supponga di avere una tabella hash di n posizioni, con le collisioni risolte per concatenazione e si supponga che n chiavi siano inserite nella tabella. Sia equamente probabile che ogni chiave corrisponda a ogni posizione. Sia M il massimo numero di chiavi in una qualunque posizione dopo che tutte le chiavi sono state inserite. Occorre adesso dimostrare che $E[M]$, il valore medio di M, ha un limite superiore $O(\lg n / \lg \lg n)$.

a. Dedurre che la probabilità Q_k che k chiavi cadano in una particolare posizione è data da

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

b. Sia P_k la probabilità che $M = k$, cioè la probabilità che la posizione contenente il maggior numero di chiavi ne contenga k. Mostrare che $P_k \le n\, Q_k$.

c. Usare l'approssimazione di Stirling (equazione (2.11)) per mostrare che $Q_k < e^k/k^k$.

d. Mostrare che esiste una costante $c > 1$ tale che $Q_{k_0} < 1/n^3$ per $k_0 = c \lg n / \lg \lg n$. Concludere che $P_k < 1/n^2$ per $k \ge k_0 = c \lg n / \lg \lg n$.

e. Mostrare che

$$E[M] \le \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \le \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n} .$$

Concludere che $E[M] = O(\lg n / \lg \lg n)$.

12-4 Scansione quadratica

Si supponga di avere una chiave k da cercare in una tabella hash con posizioni $0, 1, \ldots, m-1$, e di avere una funzione h che faccia corrispondere allo spazio delle chiavi l'insieme $\{0, 1, \ldots, m-1\}$. Lo schema della ricerca sia il seguente.

1. Calcolare il valore $i \leftarrow h(k)$, e porre $j \leftarrow 0$.
2. Esaminare la posizione i per cercare la chiave k. Se è trovata o se la posizione è vuota, terminare la ricerca.
3. Porre $j \leftarrow (j + 1) \bmod m$ e $i \leftarrow (i + j) \bmod m$ e tornare al passo 2.

Si supponga che m sia potenza di 2.

a. Mostrare che questo schema è un'istanza dello schema generale di "scansione quadratica" producendo i valori appropriati delle costanti c_1 e c_2 per l'equazione (12.5).

b. Provare che nel caso peggiore questo algoritmo esamina ogni posizione della tabella.

12-5 Hashing k-universale

Sia $\mathcal{H} = \{h\}$ una classe di funzioni hash in cui ogni h fa corrispondere l'universo U delle chiavi a $\{0, 1, \ldots, m-1\}$. Si dice che $\mathcal{H}$ è ***k-universale*** se, per ogni sequenza fissata di k chiavi distinte $\langle x_1, x_2, \ldots, x_k \rangle$ e per ogni h scelta a caso da $\mathcal{H}$, è equamente probabile che la sequenza $\langle h(x_1), h(x_2), \ldots, h(x_k) \rangle$ sia una qualunque delle m^k sequenze di lunghezza k con elementi presi da $\{0, 1, \ldots, m-1\}$.

a. Mostrare che se $\mathcal{H}$ è 2-universale, allora è universale.

b. Mostrare che la classe $\mathcal{H}$ definita nel paragrafo 12.3.3 non è 2-universale.

c. Mostrare che se si modifica la definizione di $\mathcal{H}$ nel paragrafo 12.3.3 così che ogni funzione contenga anche un termine costante b, cioè se si sostituisce $h(x)$ con

$$h_{a,b}(x) = \left(\sum_{i=0}^{r} a_i x_i + b \right) \bmod m,$$

allora $\mathcal{H}$ è 2-universale.

Note al capitolo

Knuth [123] e Gonnet [90] sono ottimi riferimenti per l'analisi di algoritmi basati sull'organizzazione hash. Knuth accredita a H. P. Luhn (1953) l'invenzione delle tabelle hash, insieme con il metodo di concatenazione per risolvere le collisioni. All'incirca nello stesso periodo, G. M. Amdahl propose l'idea dell'indirizzamento aperto.

Alberi binari di ricerca

13

Gli alberi di ricerca sono strutture di dati sulle quali vengono realizzate molte delle operazioni definite sugli insiemi dinamici, incluse SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT e DELETE. Quindi un albero di ricerca può essere usato sia come un dizionario che come una coda con priorità.

Le operazioni di base su un albero binario di ricerca richiedono un tempo proporzionale all'altezza dell'albero. Su un albero binario completo con n nodi tali operazioni sono eseguite nel caso peggiore con un tempo $\Theta(\lg n)$. Se l'albero è una catena lineare di n nodi, tuttavia, le stesse operazioni richiedono un tempo nel caso peggiore di $\Theta(n)$. Nel paragrafo 13.4 si vedrà che l'altezza di un albero binario di ricerca costruito in modo casuale è $O(\lg n)$, per cui le operazioni di base sugli insiemi dinamici richiedono tempo $\Theta(\lg n)$.

Nella pratica, non si può sempre garantire che gli alberi binari di ricerca siano costruiti in modo casuale, tuttavia vi sono varianti degli alberi binari di ricerca le cui prestazioni nel caso peggiore per le operazioni di base sono comunque buone. Il Capitolo 14 presenta una tale variante, gli RB-alberi, che hanno infatti un'altezza $O(\lg n)$. Il Capitolo 19 presenta i B-alberi che sono particolarmente adatti per la gestione di basi di dati su memoria secondaria ad accesso casuale (dischi).

Dopo la presentazione delle proprietà di base degli alberi binari di ricerca, i paragrafi seguenti mostrano come visitare un albero binario di ricerca per elencare i valori in modo ordinato, come cercare un valore in un albero binario di ricerca, come trovare il più piccolo o il più grande elemento, come trovare il successivo o il precedente di un elemento e come inserire o cancellare un elemento da un albero binario di ricerca.

Le principali proprietà matematiche degli alberi sono state presentate nel Capitolo 5.

13.1 Che cos'è un albero binario di ricerca?

Un albero binario di ricerca è organizzato, come suggerisce il nome, ad albero binario, come mostrato nella figura 13.1. Un tale albero può essere rappresentato da una struttura di dati con puntatori in cui ciascun nodo è un oggetto. Oltre ad un campo *key* e ai dati satellite, ciascun nodo contiene i campi *left*, *right* e *p* che puntano rispettivamente ai nodi corrispondenti al figlio sinistro, al figlio destro e al padre. Se un figlio o il padre manca, il corrispondente campo contiene il valore NIL. Il nodo radice è l'unico nodo dell'albero il cui campo padre è NIL.

In un albero binario di ricerca le chiavi sono sempre memorizzate in modo che sia soddisfatta la ***proprietà dell'albero binario di ricerca***:

Figura 13.1 *Alberi binari di ricerca. Per qualunque nodo x, le chiavi nel sottoalbero sinistro di x sono minori o uguali a key[x] e le chiavi nel sottoalbero destro di x sono maggiori o uguali a key[x]. Alberi binari di ricerca diversi possono rappresentare lo stesso insieme di valori. Per molte operazioni di ricerca il tempo di esecuzione nel caso peggiore è proporzionale all'altezza dell'albero.* (**a**) *Un albero binario di ricerca di altezza 2 con 6 nodi.* (**b**) *Un albero binario di ricerca meno efficiente di altezza 4 che contiene le stesse chiavi.*

Sia x un nodo di un albero binario di ricerca. Se y è un nodo del sottoalbero sinistro di x, allora $key[y] \leq key[x]$. Se y è un nodo del sottoalbero destro di x, allora $key[x] \leq key[y]$. Perciò, nella figura 13.1 (a), la chiave della radice è 5, le chiavi 2, 3 e 5 del suo sottoalbero sinistro non sono più grandi di 5, così come le chiavi 7 e 8 del suo sottoalbero destro non sono più piccole di 5. La stessa proprietà vale per tutti i nodi dell'albero. Per esempio, la chiave 3 nella figura 13.1 (a) non è più piccola della chiave 2 del suo sottoalbero sinistro e non è più grande della chiave 5 del suo sottoalbero destro.

La proprietà dell'albero binario di ricerca consente di elencare tutte le chiavi in un albero binario di ricerca in modo ordinato attraverso un semplice algoritmo ricorsivo, chiamato ***visita in ordine simmetrico***. Questo algoritmo deriva il suo nome dal fatto che la chiave della radice di un sottoalbero è elencata tra i valori del suo sottoalbero sinistro e quelli del suo sottoalbero destro. (Analogamente, una ***visita in ordine anticipato*** elenca prima la radice quindi i valori di entrambi i sottoalberi e una ***visita in ordine differito*** elenca i valori dei sottoalberi della radice e dopo la radice stessa). Per usare la seguente procedura per elencare tutti gli elementi di un albero binario di ricerca T si chiama INORDER-TREE-WALK(*root* (T)).

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
     then INORDER-TREE-WALK(left[x])
          stampa key[x]
          INORDER-TREE-WALK(right[x])
```

Per esempio la visita in ordine simmetrico elenca le chiavi di ciascuno dei due alberi binari di ricerca della figura 13.1 nell'ordine 2, 3, 5, 5, 7, 8. La correttezza dell'algoritmo deriva, per induzione, direttamente dalla proprietà dell'albero binario di ricerca. L'algoritmo richiede tempo $\Theta(n)$ per visitare un albero binario di ricerca di n nodi, dato che, dopo la chiamata iniziale, la procedura è chiamata ricorsivamente esattamente due volte per ciascun nodo nell'albero – una per il figlio sinistro e l'altra per il destro.

Esercizi

13.1-1 Disegnare alberi binari di ricerca di altezza 2, 3, 4, 5 e 6 sull'insieme di chiavi {1, 4, 5, 10, 16, 17, 21}.

13.1-2 Qual è la differenza tra la proprietà dell'albero binario di ricerca e la proprietà dello heap (7.1)? La proprietà dello heap può essere usata per elencare in modo ordinato le chiavi di un albero di n nodi in tempo $O(n)$? Giustificare la risposta.

13.1-3 Descrivere un algoritmo non ricorsivo che esegua una visita in ordine simmetrico. (*Suggerimento*: vi è una semplice soluzione che usa una pila come struttura di dati ausiliaria; vi è anche una soluzione più complicata, ma elegante che non usa la pila, che assume che si possa verificare se due puntatori sono uguali.)

13.1-4 Descrivere gli algoritmi ricorsivi che eseguono le visite anticipata e differita di un albero di n nodi in tempo $\Theta(n)$.

13.1-5 Dedurre che poiché l'ordinamento di n elementi con gli algoritmi per confronto richiede nel caso peggiore un tempo $\Omega(n \lg n)$, qualsiasi algoritmo basato su confronti usato per costruire un albero binario di ricerca a partire da una sequenza arbitraria di n elementi richiede tempo $\Omega(n \lg n)$ nel caso peggiore.

13.2 Interrogazioni su un albero binario di ricerca

L'operazione più comune eseguita su un albero binario di ricerca è la ricerca di una chiave memorizzata nell'albero. Oltre l'operazione di SEARCH, sugli alberi binari di ricerca possono essere realizzate interrogazioni come MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR. In questo paragrafo si esamineranno queste operazioni e si mostrerà che possono essere eseguite in tempo $O(h)$ su un albero binario di ricerca di altezza h.

Ricerca

La seguente procedura si usa per ricercare un nodo con una data chiave in un albero binario di ricerca. Dato un puntatore alla radice dell'albero e una chiave k, TREE-SEARCH restituisce un puntatore ad un nodo con chiave k, se esiste; altrimenti restituisce NIL.

```
TREE-SEARCH(x, k)
  if x = NIL o k = key[x]
     then return x
  if k < key[x]
     then return TREE-SEARCH(left[x], k)
     else return TREE-SEARCH(right[x], k)
```

Figura 13.2 *Interrogazioni su un albero binario di ricerca. Per cercare la chiave* 13 *nell'albero, si segue il cammino* 15 → 6 → 7 → 13 *a partire dalla radice. La più piccola chiave nell'albero è* 2, *che può essere trovata seguendo dalla radice tutti i puntatori left. La chiave più grande è* 20 *ed è trovata seguendo dalla radice tutti i puntatori right. Il successore del nodo con chiave* 15 *è il nodo con chiave* 17, *poiché questa è la chiave più piccola del sottoalbero destro del nodo* 15. *Il nodo con chiave* 13 *non ha sottoalbero destro, quindi il suo successore è il suo antenato più prossimo il cui figlio sinistro è suo antenato: in questo caso il nodo con chiave* 15 *è il suo successore.*

La procedura comincia la ricerca dalla radice e segue un cammino discendente nell'albero come mostrato nella figura 13.2. Per ogni nodo x che incontra, confronta la chiave k con $key[x]$. Se le due chiavi sono uguali la ricerca termina. Se k è più piccolo di $key[x]$, la ricerca continua nel sottoalbero sinistro di x, poiché la proprietà dell'albero binario di ricerca implica che k non si possa trovare nel sottoalbero destro. In modo simmetrico, se k è più grande di $key[x]$, la ricerca continua nel sottoalbero destro. I nodi incontrati durante la ricorsione formano un cammino che discende dalla radice dell'albero e quindi il tempo di esecuzione di TREE-SEARCH è $O(h)$ dove h è l'altezza dell'albero.

La stessa procedura può essere scritta in modo iterativo trasformando la ricorsione in un ciclo **while**. Nella maggior parte dei calcolatori questa versione è più efficiente.

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL e k ≠ key[x]
2      do if  k < key[x]
3             then x ← left[x]
4             else x ← right[x]
5  return x
```

Minimo e massimo

Un elemento dell'albero binario di ricerca la cui chiave è un minimo può sempre essere trovato seguendo i puntatori *left* dalla radice fino a che si incontra NIL come mostrato nella figura 13.2. La seguente procedura restituisce un puntatore al minimo elemento del sottoalbero con radice in un dato nodo x.

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2      do x ← left[x]
3  return x
```


La proprietà dell'albero binario di ricerca garantisce che TREE-MINIMUM sia corretta. Se un nodo x non ha il sottoalbero sinistro, allora ogni chiave del sottoalbero destro di x è grande almeno quanto $key[x]$, perciò la chiave minima del sottoalbero con radice in x è $key[x]$. Se un nodo x ha il sottoalbero sinistro, allora nessuna chiave del sottoalbero destro è più piccola di $key[x]$ ed ogni chiave del sottoalbero sinistro non è più grande di $key[x]$, perciò la chiave minima del sottoalbero con radice in x può essere trovata nel sottoalbero con radice in $left[x]$.

Lo pseudocodice per TREE-MAXIMUM è simmetrico.

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2      do x ← right[x]
3  return x
```

Poiché entrambe le procedure seguono cammini discendenti nell'albero, sono eseguite in tempo $O(h)$ su un albero di altezza h.

Successore e predecessore

Dato un nodo di un albero binario di ricerca, talvolta è importante riuscire a trovare il suo successore secondo l'ordinamento determinato dalla visita simmetrica dell'albero. Se tutte le chiavi sono distinte il successore di un nodo x è il nodo con la più piccola chiave più grande di $key[x]$. La struttura di un albero binario di ricerca consente di determinare il successore di un nodo senza dover confrontare le chiavi. La seguente procedura restituisce, se esiste, il successore di un nodo x in un albero binario di ricerca oppure NIL se x ha la chiave più grande dell'albero.

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2      then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL e x = right[y]
5      do x ← y
6          y ← p[y]
7  return y
```

Il codice di TREE-SUCCESSOR è suddiviso in due casi. Se il sottoalbero destro del nodo x non è vuoto, allora il successore di x è proprio il nodo più a sinistra del sottoalbero destro che viene trovato nella linea 2 attraverso la chiamata TREE-MINUMUM($right[x]$). Per esempio nella figura 13.2 il successore del nodo con chiave 15 è il nodo con chiave 17.

D'altra parte, se il sottoalbero destro del nodo x è vuoto ed x ha un successore y, allora y deve essere l'antenato più prossimo di x il cui figlio sinistro è anche un antenato di x. Per esempio nella figura 13.2 il successore del nodo con chiave 13 è il nodo con chiave 15. Per trovare y, si risale semplicemente l'albero da x finché si incontra un nodo che è un figlio sinistro di suo padre; ciò è gestito nelle linee 3-7 della procedura TREE-SUCCESSOR.

Il tempo di esecuzione di TREE-SUCCESSOR su un albero di altezza h è $O(h)$, infatti o si segue un cammino che risale o uno che discende l'albero. Anche la procedura TREE-PREDECESSOR, che è simmetrica alla TREE-SUCCESSOR è eseguita con un tempo $O(h)$.

In conclusione si è dimostrato il seguente teorema.

> ***Teorema 13.1***
>
> Le operazioni di un insieme dinamico SEARCH, MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR possono essere eseguite su un albero binario di ricerca di altezza h in tempo $O(h)$.

Esercizi

13-2.1 In un albero binario di ricerca, si supponga di avere numeri tra 1 e 1000 e di voler cercare il numero 363. Quale delle seguenti sequenze *non* può essere una sequenza di nodi esaminati?

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, 658, 363.

13-2.2 Il professor Volpini pensa di aver scoperto un'importante proprietà degli alberi binari di ricerca. Si supponga che la ricerca di una chiave k in un albero binario di ricerca termini in una foglia. Si considerino tre insiemi: A, le chiavi a sinistra del cammino di ricerca; B, le chiavi sul cammino di ricerca; C, le chiavi a destra del cammino di ricerca. Il professor Volpini afferma che qualunque tripla di chiavi $a \in A$, $b \in B$ e $c \in C$ deve soddisfare $a \le b \le c$. Dare il controesempio più piccolo possibile all'affermazione del professore.

13-2.3 Usare la proprietà dell'albero binario di ricerca per provare rigorosamente che il codice di TREE-SUCCESSOR è corretto.

13-2.4 La visita in ordine simmetrico di un albero binario di ricerca di n nodi può essere realizzata trovando il minimo elemento dell'albero con TREE-MINIMUM e poi eseguendo $n - 1$ chiamate a TREE-SUCCESSOR. Provare che quest'algoritmo è eseguito in tempo $\Theta(n)$.

13-2.5 Provare che k chiamate successive a TREE-SUCCESSOR in un albero binario di ricerca di altezza h richiedono tempo $O(k + h)$ a prescindere dal nodo da cui si parte.

13-2.6 Sia T un albero binario di ricerca in cui le chiavi sono tutte distinte, sia x un nodo foglia e sia y il padre di x. Mostrare che $key[y]$ o è la più piccola chiave di T più grande di $key[x]$ oppure è la più grande chiave dell'albero T più piccola di $key[x]$.

13.3 Inserzione e cancellazione

Le operazioni di inserzione e cancellazione provocano la modifica dell'insieme dinamico rappresentato con un albero binario di ricerca. La struttura di dati deve essere modificata per riflettere questo cambiamento, ma in modo tale che la proprietà dell'albero binario di ricerca continui ad essere verificata. Si vedrà che la modifica dell'albero per l'inserzione di un nuovo elemento è relativamente semplice mentre la gestione della cancellazione è in qualche modo più complessa.

Inserzione

Per inserire un nuovo valore v in un albero binario di ricerca T si usa la procedura TREE-INSERT. Alla procedura si passa un nodo z tale che $key[z] = v$, $left[z] = \text{NIL}$ e $right[z] = \text{NIL}$. La procedura modifica T e altri campi di z in modo da inserire z nella posizione appropriata nell'albero.

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4      do y ← x
 5          if  key[z] < key[x]
 6              then x ← left[x]
 7              else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10      then  root[T] ← z
11      else if  key[z] < key[y]
12              then left[y] ← z
13              else right[y] ← z
```

La figura 13.3 mostra come opera TREE-INSERT. Analogamente alle procedure TREE-SEARCH e ITERATIVE-TREE-SEARCH, la TREE-INSERT comincia dalla radice dell'albero e segue un cammino discendente. Il puntatore x segue il cammino e y punta al padre di x. Dopo l'inizializzazione, il ciclo **while** delle linee 3-7 fa discendere i due puntatori nell'albero, andando a sinistra o a destra a seconda del risultato del confronto fra $key[z]$ e $key[x]$, finché NIL non è assegnato ad x. Questo valore NIL occupa la posizione in cui si desidera collocare l'elemento z di input. Nelle linee 8-13 si assegnano i valori ai puntatori in modo tale che z sia inserito.

Come le altre operazioni primitive su alberi di ricerca la procedura TREE-INSERT quando è eseguita su un albero di altezza h richiede tempo $O(h)$.

Cancellazione

La procedura per cancellare un dato nodo z da un albero binario di ricerca prende come argomento un puntatore a z. La procedura prende in esame i tre casi mostrati nella figura 13.4.

Figura 13.3 *Inserzione di un elemento con chiave 13 in un albero binario di ricerca. I nodi grigi chiari indicano il cammino dalla radice alla posizione in cui l'elemento è inserito. La linea tratteggiata indica il collegamento aggiunto nell'albero per inserire il nuovo elemento.*

Figura 13.4 *Cancellazione di un nodo z in un albero binario di ricerca. In ciascun caso, il nodo grigio chiaro è quello effettivamente rimosso.* (a) *Se z non ha figli viene rimosso.* (b) *Se z ha un solo figlio si estrae z.* (c) *Se z ha due figli, si estrae il suo successore y, che ha al più un figlio, e quindi si sostituisce il contenuto di z con il contenuto di y.*

Se z non ha figli, si modifica suo padre $p[z]$ sostituendo il puntatore al figlio z con NIL. Se il nodo ha un unico figlio si estrae z creando un collegamento tra suo figlio e suo padre. Infine, se il nodo ha due figli, si estrae il successore y di z (che non ha figlio sinistro: si veda l'Esercizio 13.3-4) e si sostituiscono la chiave e i dati satellite di z con la chiave e i dati satellite di y.

Il codice di TREE-DELETE tratta questi tre casi in modo leggermente diverso.

```
TREE-DELETE(T, z)
 1  if left[z] = NIL o right[z] = NIL
 2      then  y ← z
 3      else  y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5      then  x ← left[y]
 6      else  x ← right[y]
 7  if x ≠ NIL
 8      then  p[x] ← p[y]
 9  if p[y] = NIL
10      then  root[T] ← x
11      else  if  y = left[p[y]]
12                then left[p[y]] ← x
13                else right[p[y]] ← x
14  if y ≠ z
15      then  key[z] ← key[y]
16            ▷ Se y ha altri campi, copia anche questi.
17  return y
```

Nelle linee 1-3 l'algoritmo determina il nodo y da estrarre. Il nodo y è il nodo di input z (se z ha al più un figlio) oppure è il successore di z (se z ha due figli). Quindi nelle linee 4-6 è assegnato ad x il figlio non NIL di y oppure NIL se y non ha figli. Il nodo y è estratto nelle linee 7-13 modificando i puntatori di $p[y]$ e di x. L'estrazione di y è in qualche modo complicata dalla necessità di un'adeguata gestione delle condizioni particolari che si verificano quando x = NIL o quando y è la radice. Infine, nelle linee 14-16, se il nodo estratto è il successore di z, il contenuto dell'oggetto z viene sostituito con quello di y. Il nodo y è restituito nella linea 17 in modo che la procedura chiamante lo possa riutilizzare nella lista libera. La procedura è eseguita in tempo $O(h)$ su un albero di altezza h.

Riassumendo, si è dimostrato il seguente teorema.

> ***Teorema 13.2***
>
> Le operazioni INSERT e DELETE su un insieme dinamico possono essere eseguite in tempo $O(h)$ utilizzando un albero binario di ricerca di altezza h.

Esercizi

13.3-1 Fornire una versione ricorsiva della procedura TREE-INSERT.

13.3-2 Si supponga che un albero binario di ricerca sia costruito attraverso una sequenza di inserzioni di valori distinti. Dedurre che il numero di nodi esaminati per la ricerca di un valore è uno più il numero di nodi esaminati quando quel valore è stato inserito nell'albero.

13.3-3 Si può ordinare un dato insieme di n numeri costruendo prima un albero binario di ricerca che contenga questi numeri (usando una sequenza di TREE-INSERT per inserire i numeri uno alla volta) e elencando successivamente i numeri con una visita in ordine simmetrico dell'albero. Quali sono i tempi di esecuzione nel caso migliore e nel caso peggiore di questo algoritmo di ordinamento?

13.3-4 Mostrare che, in un albero binario di ricerca, se un nodo ha due figli, allora il suo successore non ha figlio sinistro e il suo predecessore non ha figlio destro.

13.3-5 Si supponga che un'altra struttura di dati contenga un puntatore al nodo y di un albero binario di ricerca e si supponga che il predecessore z di y sia cancellato dall'albero con la procedura TREE-DELETE. Quale problema si può manifestare? Come può essere riscritta TREE-DELETE per risolvere questo problema?

13.3-6 L'operazione di cancellazione è "commutativa" nel senso che la cancellazione di x e poi di y da un albero binario di ricerca produce lo stesso albero che si ottiene cancellando prima y e poi x? Se sì spiegare perché; se no dare un controesempio.

13.3-7 Quando nella TREE-DELETE il nodo z ha due figli, si potrebbe estrarre il suo predecessore piuttosto che il suo successore. Una strategia equilibrata, che dia la stessa priorità al predecessore e al successore, empiricamente produce migliori prestazioni. Come si potrebbe cambiare TREE-DELETE in modo da realizzare tale strategia?

* 13.4 Alberi binari di ricerca costruiti in modo casuale

Abbiamo mostrato che tutte le operazioni di base su un albero binario di ricerca richiedono tempo $O(h)$, dove h è l'altezza dell'albero. L'altezza di un albero binario di ricerca, però, varia a seconda di come gli elementi sono inseriti e cancellati. Per analizzare il comportamento di un albero binario di ricerca in pratica, è ragionevole fare delle ipotesi statistiche sulla distribuzione delle chiavi e sulla sequenza di inserzioni e cancellazioni.

Sfortunatamente, si conosce poco dell'altezza media di un albero binario di ricerca quando lo si crea usando inserzioni e cancellazioni. Quando l'albero è creato solo attraverso inserzioni l'analisi diventa più semplice da trattare. Si definisce perciò un ***albero binario di ricerca costruito in modo casuale*** su n chiavi distinte quello che deriva dall'inserzione delle chiavi in ordine casuale in un albero inizialmente vuoto, dove ciascuna delle $n!$ permutazioni delle chiavi di input è equiprobabile. (L'Esercizio 13.4-2 chiederà di dimostrare come questa nozione sia diversa dall'assumere che ogni albero binario di ricerca su n chiavi sia equiprobabile). Scopo di questo paragrafo è di mostrare che l'altezza media di un albero binario di ricerca costruito in modo casuale su n chiavi è $O(\lg n)$.

Si comincia analizzando la struttura degli alberi binari di ricerca che sono costruiti solo attraverso inserzioni.

Lemma 13.3

Sia T l'albero che risulta dall'inserzione di n chiavi distinte $k_1, k_2, \ldots, k_n$ (nell'ordine) in un albero binario di ricerca inizialmente vuoto. Allora k_i è un antenato di k_j in T, per $1 \le i \le j \le n$, se e solo se

$$k_i = \min\{k_l : 1 \le l \le i \text{ e } k_l > k_j\}$$

oppure

$$k_i = \max\{k_l : 1 \le l \le i \text{ e } k_l < k_j\}.$$

Dimostrazione. $\Rightarrow$ Si supponga che k_i sia un antenato di k_j. Si consideri l'albero T_i che risulta dopo l'inserzione delle chiavi $k_1, k_2, \ldots, k_i$. In T_i il cammino dalla radice a k_i è uguale al cammino in T dalla radice a k_i. Quindi, se k_j fosse stato inserito in T_i sarebbe diventato il figlio destro o sinistro di k_i. Di conseguenza (si veda l'Esercizio 13.2-6), k_i è la più piccola chiave tra $k_1, k_2, \ldots, k_i$ che è più grande di k_j oppure è la più grande chiave tra $k_1, k_2, \ldots, k_i$ che è più piccola di k_j.

$\Leftarrow$ Si supponga che k_i sia la più piccola chiave tra $k_1, k_2, \ldots, k_i$ che è più grande di k_j. (Il caso in cui k_i sia la più grande chiave tra $k_1, k_2, \ldots, k_i$ che è più piccola di k_j è gestito in modo simmetrico.) Il confronto di k_j con qualsiasi chiave sul cammino in T dalla radice a k_i produce gli stessi risultati del confronto di k_i con le stesse chiavi. Quindi, quando k_j è inserita, si segue un cammino che attraversa k_i ed essa è inserita come discendente di k_i. ■

Come conseguenza del Lemma 13.3, si può caratterizzare precisamente la profondità di una chiave in base alla permutazione dell'input.

Corollario 13.4

Sia T l'albero risultante dall'inserzione di n chiavi distinte $k_1, k_2, \ldots, k_n$ (nell'ordine) in un albero binario di ricerca inizialmente vuoto. Per una data chiave k_j, dove $1 \le j \le n$, si definisce

$$G_j = \{k_i : 1 \le i < j \text{ e } k_l > k_i > k_j \text{ per ogni } l < i \text{ tale che } k_l > k_j\}$$

e

$$L_j = \{k_i : 1 \le i < j \text{ e } k_l < k_i < k_j \text{ per ogni } l < i \text{ tale che } k_l < k_j\}.$$

Allora le chiavi sul cammino dalla radice a k_j sono esattamente le chiavi in $G_j \cup L_j$, e la profondità in T di qualunque chiave k_i è

$$d(k_j, T) = |G_j| + |L_j|.$$ ■

La figura 13.5 illustra i due insiemi G_j e L_j. L'insieme G_j contiene qualunque chiave k_i inserita prima di k_j tale che k_i è la più piccola chiave tra $k_1, k_2, \ldots, k_i$ che è più grande di k_j. La struttura di L_j è simmetrica. Per comprendere meglio l'insieme G_j, vediamo un metodo con cui si possono enumerare i suoi elementi. Tra le chiavi $k_1, k_2, \ldots, k_{j-1}$ si considerino in ordine quelle più grandi di k_j. Queste chiavi sono mostrate nella figura 13.5 come G'_j. Man mano che ogni chiave è considerata si mantiene il minimo corrente. L'insieme G_j consiste di quegli elementi che aggiornano il minimo corrente.

(a)

chiavi	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 *I due insiemi G_j e L_j che contengono le chiavi su un cammino dalla radice di un albero binario di ricerca fino alla chiave $k_j = 17$.* (a) *I nodi con chiavi in G_j sono neri e i nodi con chiavi in L_j sono bianchi. Tutti gli altri nodi sono grigi. Il cammino dalla radice al nodo con chiave k_j è grigio. Le chiavi alla sinistra della linea tratteggiata sono minori di k_j, mentre quelle alla sua destra sono maggiori. L'albero è costruito tramite l'inserzione delle chiavi mostrate nella lista superiore di* (b). *L'insieme $G'_j = \{21, 25, 19, 29\}$ consiste degli elementi che sono stati inseriti prima di 17 e che sono più grandi di 17. L'insieme $G_j = \{21, 19\}$ consiste degli elementi che aggiornano il minimo corrente dell'insieme G'_j. Quindi, la chiave 21 è in G_j, poiché è il primo elemento. La chiave 25 non è in G_j, essendo più grande del minimo corrente 21. La chiave 19 è in G_j, essendo più piccola del minimo corrente 21. La chiave 29 non è in G_j, essendo più grande del minimo corrente 19. Le strutture di L'_j e di L_j sono simmetriche.*

Al fine dell'analisi si può semplificare in qualche modo questo scenario. Si supponga che n numeri distinti siano inseriti uno alla volta in un insieme dinamico. Se tutte le permutazioni dei numeri sono equiprobabili, quante volte cambia, in media, il minimo dell'insieme? Per rispondere a questa domanda si supponga che l'i-esimo numero inserito sia k_i, per $i = 1, 2, \ldots, n$. La probabilità che k_i sia il minimo dei primi i numeri è $1/i$, poiché il rango di k_i tra i primi i numeri è equiprobabile che sia uno qualunque degli i possibili ranghi. Di conseguenza il numero medio di cambiamenti per il minimo dell'insieme è

$$\sum_{i=1}^{n} \frac{1}{i} = H_n ,$$

dove $H_n = \ln n + O(1)$ è l'n-esimo numero armonico (si veda l'Equazione (3.5) e il Problema 6-2).

Ci si aspetta dunque che il numero di cambiamenti del minimo sia approssimativamente $\ln n$; il seguente lemma mostra che è minima la probabilità che sia molto più grande.

Lemma 13.5

Sia $k_1, k_2, \ldots, k_n$ una permutazione casuale di n numeri distinti, e sia S la variabile casuale che rappresenta la cardinalità dell'insieme

$$S = \{k_i : 1 \le i \le n \text{ e } k_l > k_i \text{ per ogni } l < i\}. \tag{13.1}$$

Allora $\Pr\{|S| \ge (\beta + 1) H_n\} \le 1/n^2$, dove H_n è l'n-esimo numero armonico e $\beta = 4.32$ soddisfa l'equazione $(\ln\beta - 1)\beta = 2$.

Dimostrazione. Si può determinare la cardinalità dell'insieme S tramite n prove di Bernoulli, dove un successo si verifica nell'i-esima prova se k_i è più piccolo degli elementi $k_1, k_2, \ldots, k_{i-1}$. Un successo nell'$i$-esima prova si verifica con probabilità $1/i$. Le prove sono indipendenti, poiché la probabilità che k_i sia il minimo tra $k_1, k_2, \ldots, k_i$ è indipendente da come sono ordinati $k_1, k_2, \ldots, k_{i-1}$.

Si può usare il Teorema 6.6 per dare un limite alla probabilità che $|S| \ge (\beta + 1) H_n$. Il valore atteso di $|S|$ è $\mu = H_n \ge \ln n$. Poiché $\beta > 1$, per il Teorema 6.6 vale

$$\begin{aligned}
\Pr\{|S| \ge (\beta + 1)H_n\} &= \Pr\{|S| - \mu \ge \beta H_n\} \\
&\le \left(\frac{eH_n}{\beta H_n}\right)^{\beta H_n} \\
&= e^{(1-\ln\beta)\beta H_n} \\
&\le e^{-(\ln\beta - 1)\beta \ln n} \\
&= n^{-(\ln\beta - 1)\beta} \\
&= 1/n^2 ,
\end{aligned}$$

che segue dalla definizione di β. ■

Si hanno ora gli strumenti per limitare l'altezza di un albero binario di ricerca costruito in modo casuale.

Teorema 13.6

L'altezza media di un albero binario di ricerca costruito in modo casuale con n chiavi distinte è $O(\lg n)$.

Dimostrazione. Sia $k_1, k_2, \ldots, k_n$ una permutazione casuale delle n chiavi e sia T l'albero binario di ricerca risultante dopo aver inserito le chiavi nell'ordine in un albero inizialmente vuoto. Si cominci col considerare la probabilità che la profondità $d(k_i, T)$ di una data chiave k_i sia almeno t, dove t è un valore arbitrario. Dalla caratterizzazione di $d(k_i, T)$ del Corollario 13.4, se la profondità di k_i è almeno t, allora la cardinalità di uno dei due insiemi G_i e L_i deve essere almeno $t/2$. Per cui

$$\Pr\{d(k_i, T) \ge t\} \le \Pr\{|G_i| \ge t/2\} + \Pr\{|L_i| \ge t/2\}. \tag{13.2}$$

Esaminiamo prima $\Pr\{|G_i| \ge t/2\}$. Si ha

$$\Pr\{|G_i| \ge t/2\} = \Pr\{|\{k_l : 1 \le l < j \text{ e } k_l > k_i > k_j \text{ per ogni } l < i\}| \ge t/2\}$$

$$\leq \Pr\{|\{k_i : i \leq n \text{ e } k_i > k_j \text{ per ogni } l < i\}| \geq t/2\}$$
$$= \Pr\{|S| \geq t/2\},$$

dove S è definita come nell'equazione (13.1). Per giustificare questo ragionamento, si noti che la probabilità non decresce se si estende il rango di i da $i < j$ ad $i \leq n$, poiché all'insieme sono aggiunti più elementi. Analogamente, la probabilità non decresce se si rimuove la condizione che $k_i > k_j$, poiché si sta sostituendo una permutazione casuale di un numero di elementi che può essere minore di n (quei k_i che sono più grandi di k_j) con una permutazione casuale di n elementi.

Usando un ragionamento simmetrico, si può provare che

$$\Pr\{|L_j| \geq t/2\} \leq \Pr\{|S| \geq t/2\},$$

per cui, dalla disuguaglianza (13.2), si ottiene

$$\Pr\{d(k_j, T) \geq t\} \leq 2\Pr\{|S| \geq t/2\}.$$

Se si sceglie $t = 2(\beta + 1)H_n$, dove H_n è l'n-esimo numero armonico e $\beta \approx 4.32$ soddisfa $(\ln\beta - 1)\beta = 2$, si può applicare il Lemma 13.5 per concludere che

$$\Pr\{d(k_j, T) \geq 2(\beta + 1) H_n\} \leq 2\Pr\{|S| \geq (\beta + 1) H_n\}$$
$$\leq 2/n^2.$$

Poiché vi sono al più n nodi in un albero binario di ricerca costruito in modo casuale, la probabilità che la profondità di *qualunque* nodo sia almeno $2(\beta + 1)H_n$ è dunque, dalla disuguaglianza di Boole (6.22), al più $n(2/n^2) = 2/n$. Quindi, per una frazione pari ad almeno $1 - 2/n$ delle volte, l'altezza di un albero binario di ricerca costruito in modo casuale è minore di $2(\beta + 1)H_n$, e al più $2/n$ volte l'altezza non supera n. L'altezza media è allora al più $(2(\beta + 1)H_n)(1 - 2/n) + n(2/n) = O(\lg n)$. ■

Esercizi

13.4-1 Descrivere un albero binario di ricerca di n nodi tale che la profondità media di un nodo nell'albero sia $\Theta(\lg n)$ ma l'altezza dell'albero sia $\omega(\lg n)$. Quanto può essere grande l'altezza di un albero binario di ricerca di n nodi se la profondità media di un nodo è $\Theta(\lg n)$?

13.4-2 Mostrare che la nozione di albero binario di ricerca di n chiavi scelto in modo casuale, dove ogni albero binario di ricerca di n chiavi può essere scelto in modo equiprobabile, è diversa dalla nozione data in questo paragrafo di albero binario di ricerca costruito in modo casuale. (*Suggerimento*: elencare le possibilità per $n = 3$.)

* *13.4-3* Data una costante $r \geq 1$, si determini t in modo che la probabilità che l'altezza di un albero binario di ricerca costruito in modo casuale sia almeno tH_n risulti minore di $1/n^r$.

* *13.4-4* Si consideri l'algoritmo RANDOMIZED-QUICKSORT che opera su una sequenza di input di n numeri. Provare che per qualunque costante $k > 0$, per tutte le $n!$ permutazioni dell'input, tranne che per $O(1/n^k)$ permutazioni, il tempo di esecuzione è $O(n \lg n)$.

Problemi

13-1 Alberi binari di ricerca con chiavi uguali

La presenza di chiavi uguali pone qualche problema nella realizzazione degli alberi binari di ricerca.

a. Quali sono le prestazioni asintotiche di TREE-INSERT quando viene usata per inserire n elementi con chiavi identiche in un albero binario di ricerca inizialmente vuoto?

Ci si propone di migliorare TREE-INSERT verificando prima della linea 5 se $key[z] = key[x]$ e verificando prima della linea 11 se $key[z] = key[y]$. Se vale l'uguaglianza si segue una delle seguenti strategie. Per ogni strategia, si trovino le prestazioni asintotiche dell'inserzione di n elementi con chiavi identiche in un albero binario di ricerca inizialmente vuoto. (Le strategie sono descritte per la linea 5 in cui si confrontano le chiavi di z e x. Per applicare la stessa strategia alla linea 11, si dovrà sostituire y ad x.)

b. Mantenere una variabile booleana $b[x]$ sul nodo x e assegnare ad x $left[x]$ oppure $right[x]$ in base al valore di $b[x]$, che si alterna tra FALSE e TRUE ogni volta che il nodo è visitato durante la TREE-INSERT.

c. Mantenere una lista di nodi con chiavi uguali ad x ed inserire z nella lista.

d. Assegnare in modo casuale ad x $left[x]$ oppure $right[x]$. (Fornire le prestazioni nel caso peggiore e derivare informalmente quelle nel caso medio.)

13-2 Alberi lessicografici

Date due stringhe $a = a_0a_1 \ldots a_p$ e $b = b_0b_1 \ldots b_q$, dove ogni a_i ed ogni b_j appartengono ad un insieme ordinato di caratteri, si dice che la stringa a è ***lessicograficamente minore*** della stringa b se vale una delle due seguenti affermazioni:

1. esiste un intero j, con $0 \leq j \leq \min(p, q)$, tale che $a_i = b_i$ per ogni $i = 0, 1, \ldots, j - 1$ e $a_j < b_j$,
2. $p < q$ e $a_i = b_i$ per ogni $i = 0, 1, \ldots, p$.

Per esempio, se a e b sono stringhe di cifre binarie, allora $10100 < 10110$ dalla regola 1 (per $j = 3$) e $10100 < 101000$ per la regola 2. L'ordinamento è simile a quello usato nei vocabolari.

La struttura di dati ***albero lessicografico*** mostrata nella figura 13.6 memorizza le stringhe binarie 1011, 10, 011, 100 e 0. Quando si cerca la chiave $a = a_0a_1 \ldots a_p$ dal nodo di profondità i si va a sinistra se $a_i = 0$ e a destra se $a_i = 1$. Sia S un insieme di stringhe binarie distinte le cui lunghezze hanno come somma n: mostrare come usare un albero lessicografico per ordinare lessicograficamente S in tempo $\Theta(n)$. Per l'esempio della figura 13.6, l'output dell'ordinamento dovrebbe essere la sequenza: 0, 011, 10, 100, 1011.

13-3 Profondità media di un nodo in un albero binario di ricerca costruito in modo casuale

In questo problema si prova che la profondità media di un nodo in un albero binario di ricerca di n nodi costruito in modo casuale è $O(\lg n)$. Sebbene questo risultato sia più debole di quello del Teorema 13.6, la tecnica usata rivela una somiglianza sorprendente tra la costruzione di un albero binario di ricerca e l'esecuzione di RANDOMIZED-QUICKSORT del paragrafo 8.3.

Figura 13.6 *Un albero lessicografico che memorizza le stringhe binarie* 1011, 10, 011, 100 *e* 0. *La chiave di ogni nodo può essere determinata attraversando il cammino dalla radice a quel nodo. Non è necessario, quindi, memorizzare le chiavi nei nodi; nella figura sono mostrate le chiavi solo per maggiore chiarezza. I nodi sono scuri se le chiavi corrispondenti non sono nell'albero; tali nodi sono presenti solo per tracciare il cammino per altri nodi.*

Cominciamo ricordando dal Capitolo 5 che la lunghezza del cammino interno $P(T)$ di un albero binario T è la somma delle profondità di tutti i nodi x di T, profondità che si denota con $d(x, T)$.

a. Dedurre che la profondità media di un nodo in T è

$$\frac{1}{n}\sum_{x \in T} d(x, T) = \frac{1}{n}P(T) .$$

Pertanto, si deve mostrare che il valore medio di $P(T)$ è di $O(n \lg n)$.

b. Siano rispettivamente T_L e T_R i sottoalberi sinistro e destro dell'albero T. Dedurre che se T ha n nodi, allora

$$P(T) = P(T_L) + P(T_R) + n - 1 .$$

c. Sia $P(n)$ la lunghezza media del cammino interno di un albero binario di ricerca di n nodi costruito in modo casuale. Mostrare che

$$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1) .$$

d. Mostrare che $P(n)$ può essere riscritta come

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

e. Facendo riferimento all'analisi della versione randomizzata di quicksort, concludere che $P(n) = O(n \lg n)$.

Ad ogni chiamata ricorsiva di quicksort si sceglie casualmente un elemento perno per partizionare l'insieme di elementi che devono essere ordinati. Ogni nodo di un albero binario di ricerca partiziona l'insieme di elementi che ricadono nel sottoalbero radicato in esso.

f. Descrivere una realizzazione del quicksort in cui i confronti per ordinare un insieme di elementi sono esattamente gli stessi di quelli necessari ad inserire gli elementi in un albero binario di ricerca. (L'ordine in cui i confronti sono fatti può essere differente, ma i confronti devono essere gli stessi.)

13-4 Il numero degli alberi binari diversi con n nodi

Sia b_n *il* numero di alberi binari diversi con n nodi. In questo problema bisogna trovare una formula per b_n e una stima asintotica.

a. Mostrare che $b_0 = 1$ e che, per $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k} .$$

b. Sia $B(x)$ la funzione generatrice

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

(si veda il Problema 4-6 per la definizione di funzione generatrice). Mostrare che $B(x) = xB(x)^2 + 1$ e quindi

$$B(x) = \frac{1}{2x}\left(1 - \sqrt{1 - 4x}\right) .$$

Lo ***sviluppo in serie di Taylor*** di $f(x)$ nel punto iniziale $x = a$ è dato da

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k ,$$

dove $f^{(k)}(x)$ è la derivata di ordine k di f in x.

c. Mostrare che

$$b_n = \frac{1}{n+1}\binom{2n}{n} .$$

(l'n-esimo ***numero Catalano***) usando lo sviluppo in serie di Taylor di $\sqrt{1 - 4x}$ nel punto iniziale $x = 0$.

(Se si vuole, invece di usare lo sviluppo in serie di Taylor, si può usare la generalizzazione dello sviluppo in serie binomiale (6.5) ad esponenti n non interi, dove per qualunque numero reale n e qualunque intero k, si interpreta $\binom{n}{k}$ come $n(n-1)\ldots(n-k+1)/k!$ se $k \geq 0$, 0 altrimenti.)

d. Mostrare che

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}(1 + O(1/n)) .$$

Note al capitolo

Knuth [123] presenta un'ampia dissertazione su alberi binari di ricerca semplici e molte loro variazioni. Sembra che gli alberi binari di ricerca siano stati inventati da varie persone in modo indipendente verso la fine degli anni '50.

RB-alberi

14

Nel Capitolo 13 si è mostrato come un albero binario di ricerca di altezza h possa essere utilizzato per realizzare le operazioni di base su un insieme dinamico – come SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT e DELETE – in tempo $O(h)$. Perciò queste operazioni sono veloci se l'altezza dell'albero è piccola; ma se l'altezza è grande allora le loro prestazioni diventano paragonabili a quelle ottenute con una lista concatenata. Gli RB-alberi rappresentano uno dei molti schemi di alberi di ricerca "bilanciati" che consentono di eseguire le operazioni di base degli insiemi dinamici in tempo $O(\lg n)$ nel caso peggiore.

14.1 Proprietà degli RB-alberi

Un ***RB-albero*** è un albero binario di ricerca con in più un campo binario in ogni nodo, il suo ***colore***, che può essere RED (rosso) oppure BLACK (nero). Attraverso precise regole di colorazione dei nodi su qualsiasi cammino dalla radice a una foglia, si ottiene che nessun cammino di un RB-albero risulti lungo più del doppio di qualsiasi altro: quindi l'albero è approssimativamente ***bilanciato***.

Ciascun nodo dell'albero contiene i campi *color*, *key*, *left*, *right* e *p*. Se un figlio o il padre di un nodo non esistono, allora il corrispondente campo puntatore vale NIL. Questi valori NIL sono considerati come puntatori a nodi esterni (foglie) dell'albero binario di ricerca, mentre per i nodi interni dell'albero è mantenuta la normale impostazione basata sulla chiave.

Un albero binario di ricerca è un RB-albero se soddisfa le seguenti ***proprietà RB***:

1. Ciascun nodo è rosso o nero.
2. Ciascuna foglia (NIL) è nera.
3. Se un nodo è rosso allora entrambi i figli sono neri.
4. Ogni cammino semplice da un nodo ad una foglia sua discendente contiene lo stesso numero di nodi neri.

Un esempio di RB-albero è mostrato nella figura 14.1.

Il numero di nodi neri su un cammino da un nodo x, non incluso, ad una foglia è chiamato ***b-altezza*** del nodo ed è indicata con $bh(x)$. Per la proprietà 4, la nozione di b-altezza è ben definita dato che tutti i cammini discendenti dal nodo hanno il medesimo numero di nodi neri. Si definisce la b-altezza di un RB-albero come la b-altezza della sua radice.

Il seguente lemma mostra per quale motivo gli RB-alberi risultano essere buoni alberi di ricerca.

Figura 14.1 *Un albero RB con i nodi neri in nero e quelli rossi in grigio. Ogni nodo in un albero RB è rosso o nero, ogni foglia* (NIL) *è nera; i figli di un nodo rosso sono entrambi neri ed ogni cammino semplice da un nodo ad una foglia che è sua discendente contiene sempre lo stesso numero di nodi neri. Ogni nodo con* NIL *è etichettato con la sua b-altezza; i nodi* NIL *hanno una b-altezza uguale a* 0.

Lemma 14.1

Un RB-albero con n nodi interni ha, al più, un'altezza di $2 \lg(n + 1)$.

Dimostrazione. Per prima cosa mostriamo che il sottoalbero radicato in un qualsiasi nodo x contiene almeno $2^{bh(x)} - 1$ nodi interni. Dimostriamo questo risultato per induzione sull'altezza di x. Se l'altezza di x è 0 allora x deve essere una foglia (NIL) e chiaramente il sottoalbero radicato in x contiene almeno $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nodi interni. Per il passo di induzione si può considerare un nodo x che abbia un'altezza positiva e sia un nodo interno con due figli. Ciascun figlio ha una b-altezza di $bh(x)$ o $bh(x) - 1$ a seconda che il suo colore sia rispettivamente rosso o nero. Dato che l'altezza di ciascun figlio di x è inferiore all'altezza di x stesso, si può applicare l'ipotesi induttiva per concludere che ogni figlio ha almeno $2^{bh(x)-1} - 1$ nodi interni. Quindi il sottoalbero radicato in x contiene almeno $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodi interni, dimostrando così il risultato che si voleva ottenere.

Per completare la dimostrazione del lemma, sia h l'altezza dell'albero. Per la proprietà 3 almeno metà nodi su qualsiasi cammino semplice dalla radice ad una foglia, esclusa la radice, devono essere neri. Di conseguenza la b-altezza della radice deve essere almeno $h/2$; per cui

$n \geq 2^{h/2} - 1$.

Portando l'1 a sinistra della disuguaglianza ed applicando il logaritmo ad entrambi i lati si ottiene $\lg(n + 1) \geq h/2$ o $h \leq 2 \lg(n + 1)$. ■

Una conseguenza immediata di questo lemma è che le operazioni degli insiemi dinamici, SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM e MAXIMUM possono essere realizzate su un RB-albero ottenendo tempi di esecuzione $O(\lg n)$, dato che su un comune albero di ricerca di altezza h i loro tempi di esecuzione sono $O(h)$ (come mostrato nel Capitolo 13) ed un qualsiasi RB-albero di n nodi è un albero di ricerca di altezza $O(\lg n)$. Benché le operazioni TREE-INSERT e TREE-DELETE del Capitolo 13 siano eseguite in tempo $O(\lg n)$ quando sono applicate ad un RB-albero di input, esse non producono i risultati teorici previsti per le operazioni di INSERT e DELETE di un insieme dinamico, poiché non garantiscono che l'albero di ricerca da loro modificato rimanga ancora un RB-albero. Tuttavia, nei paragrafi 14.3 e 14.4 si vedrà come anche queste due ultime operazioni possano essere eseguite su un RB-albero in tempo $O(\lg n)$.

Esercizi

14.1-1 Disegnare l'albero binario di ricerca completo di altezza 3 contenente le chiavi $\{1, 2, \ldots, 15\}$. Aggiungere le foglie NIL e fornire tre diverse colorazioni dei nodi in modo che i tre risultanti RB-alberi abbiano rispettivamente b-altezza uguale a: 2, 3 e 4.

14.1-2 Si supponga che la radice di un RB-albero sia rossa. Se la si colora di nero, l'albero è ancora un RB-albero?

14.1-3 Mostrare che in un RB-albero il cammino semplice più lungo da un nodo x ad una foglia sua discendente ha una lunghezza al più doppia di quella del cammino semplice più corto dal nodo x ad una foglia sua discendente.

14.1-4 Qual è il più grande numero possibile di nodi interni in un RB-albero con b-altezza uguale a k? Qual è il più piccolo numero possibile?

14.1-5 Descrivere un RB-albero con n nodi che realizzi il rapporto più alto tra i nodi interni rossi e quelli interni neri. Qual è questo rapporto? Quale albero ha il più basso rapporto possibile e qual è questo rapporto?

14.2 Rotazioni

Le operazioni TREE-INSERT e TREE-DELETE sugli alberi binari di ricerca impiegano tempo $O(\lg n)$ se eseguite su un RB-albero di n chiavi. Dato che esse modificano la struttura è possibile che l'albero prodotto violi le proprietà degli RB-alberi che sono state elencate nel paragrafo 14.1. Per ripristinare queste proprietà è necessario cambiare il colore a qualche nodo dell'albero e modificare la struttura dei puntatori.

L'operazione che cambia la struttura dei puntatori prende il nome di ***rotazione***: è un'operazione locale dell'albero di ricerca che non modifica l'ordinamento delle chiavi secondo la visita in ordine simmetrico. La figura 14.2 mostra i due tipi di rotazione: la rotazione sinistra e quella destra. Quando si esegue una rotazione sinistra su un nodo x, si

Figura 14.2 *Le operazioni di rotazione su un albero binario di ricerca. L'operazione* RIGHT-ROTATE(T, y) *trasforma la configurazione dei due nodi sulla sinistra nella configurazione che è sulla destra cambiando un numero costante di puntatori. La configurazione di destra può essere trasformata nella configurazione di sinistra attraverso l'operazione inversa* LEFT-ROTATE(T, x). *I due nodi potrebbero essere ovunque in un albero binario di ricerca. Le lettere α, β e γ rappresentano sottoalberi arbitrari. Un'operazione di rotazione mantiene l'ordinamento delle chiavi secondo la visita in ordine simmetrico dell'albero: le chiavi in α precedono key[x], che precede le chiavi in β, che precede key[y], che precede le chiavi in γ.*

assume che il figlio destro y non sia NIL. La rotazione sinistra fa perno sull'arco da x a y. Essa pone y come nuova radice del sottoalbero, x come figlio sinistro di y ed il figlio sinistro di y come nuovo figlio destro di x.

La procedura LEFT-ROTATE assume che $right[x] \neq$ NIL.

```
LEFT-ROTATE(T, x)
y ← right[x]                    ▷ Inizializzazione di y.
right[x] ← left[y]              ▷ Rotazione del sottoalbero sinistro di y
                                ▷ nel sottoalbero destro di x.
if left[y] ≠ NIL
    then p[left[y]] ← x
p[y] ← p[x]                     ▷ Collegamento del padre di x ad y.
if p[x] = NIL
    then root[T] ← y
    else if x = left[p[x]]
            then left[p[x]] ← y
            else right[p[x]] ← y
left[y] ← x                     ▷ Spostamento di x alla sinistra di y.
p[x] ← y
```

La figura 14.3 illustra il funzionamento della procedura LEFT-ROTATE. Il codice per la procedura RIGHT-ROTATE è simile. Entrambe le procedure sono eseguite in tempo $O(1)$; esse operano solo sui puntatori e lasciano inalterati tutti gli altri campi di un nodo.

Esercizi

14.2-1 Disegnare l'RB-albero che risulta dopo che è stata eseguita la procedura TREE-INSERT con chiave 36 sull'albero della figura 14.1. Se si colora di rosso il nodo inserito, si ottiene ancora un RB-albero? E se lo si colora di nero?

14.2-2 Scrivere lo pseudocodice per la procedura RIGHT-ROTATE.

14.2-3 Siano a, b e c nodi arbitrari rispettivamente dei sottoalberi α, β e γ dell'albero di sinistra della figura 14.2. Come cambiano le profondità di a, b e c quando è eseguita la rotazione destra sul nodo y della figura?

14.2-4 Mostrare che qualsiasi albero di n nodi può essere trasformato in un qualsiasi altro albero di n nodi usando $O(n)$ rotazioni. (*Suggerimento*: mostrare prima che sono sufficienti al più $n-1$ rotazioni destre per trasformare qualsiasi albero in un albero binario in cui ogni nodo ha solo un figlio destro).

Figura 14.3 *Un esempio di come la procedura* LEFT-ROTATE(T, x) *modifichi un albero binario di ricerca. Le foglie* NIL *sono omesse. Le visite in ordine simmetrico dell'albero in input e di quello modificato producono la stessa sequenza di valori delle chiavi.*

14.3 Inserzione

L'inserzione di un nodo in un RB-albero di n nodi può essere eseguita in un tempo $O(\lg n)$. Si usa la procedura TREE-INSERT (paragrafo 13.3) per inserire un nodo x in un albero T come se fosse un comune albero binario di ricerca, poi il nodo inserito si colora di rosso. Per garantire che le proprietà di un RB-albero siano mantenute, si deve poi sistemare l'albero modificato ricolorando i nodi ed eseguendo le rotazioni. La maggior parte del codice della procedura RB-INSERT serve per gestire i vari casi che si possono presentare durante la fase di risistemazione dell'albero.

```
RB-INSERT(T, x)
TREE-INSERT(T, x)
color[x] ← RED
while x ≠ root[T] e color[p[x]] = RED
    do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]
                if color[y] = RED
                    then color[p[x]] ← BLACK          ▷ caso 1
                         color[y] ← BLACK             ▷ caso 1
                         color[p[p[x]]] ← RED         ▷ caso 1
                         x ← p[p[x]]                  ▷ caso 1
```

```
11          else if x = right[p[x]]
12                  then x ← p[x]                          ▷ caso 2
13                       LEFT-ROTATE(T, x)                 ▷ caso 2
14                  color[p[x]] ← BLACK                    ▷ caso 3
15                  color[p[p[x]]] ← RED                   ▷ caso 3
16                  RIGHT-ROTATE(T, p[p[x]])               ▷ caso 3
17      else (analogo al ramo then con "right" e "left" scambiati)
18  color[root[T]] ← BLACK
```

Il codice di RB-INSERT è meno complesso di quanto sembri; sarà esaminato suddividendolo in tre parti principali. Per prima cosa si devono determinare quali violazioni alle proprietà dell'RB-albero vengono apportate dopo che il nodo x è stato inserito e colorato di rosso nelle linee 1-2. In secondo luogo si esaminerà il risultato globale del ciclo **while** nelle linee 3-17. Infine, come terzo punto, si spiegheranno i tre casi in cui è suddiviso il ciclo del **while** e si vedrà come si comportano per raggiungere il loro obiettivo. La figura 14.4 mostra il comportamento della procedura RB-INSERT su di un esempio.

Quali sono le proprietà di un RB-albero che possono essere violate nelle linee 1-2? La proprietà 1 e la proprietà 2 continuano ad essere soddisfatte dato che il nuovo nodo inserito è rosso e ha figli NIL. Anche la proprietà 4, che dice che il numero di nodi neri è sempre lo stesso per qualsiasi cammino da un dato nodo, è soddisfatta in quanto il nodo x sostituisce un nodo NIL (nero), viene colorato di rosso e ha figli NIL. Perciò l'unica proprietà che potrebbe essere violata è la proprietà 3, che dice che un nodo rosso non può avere figli rossi; più precisamente questa proprietà è violata se il padre di x è rosso, dato che x stesso viene colorato di rosso nella linea 2. La figura 14.4(a) mostra un esempio di tale violazione quando si inserisce il nodo x.

Lo scopo del ciclo **while** nelle linee 3-17 è quello di far "risalire" nell'albero questa violazione, mentre la proprietà 4 rimane sempre valida. Infatti, all'inizio di ciascuna iterazione, x punta ad un nodo rosso con padre rosso: l'unica violazione alle proprietà dell'albero. Per ogni iterazione ci sono due possibili uscite: il puntatore x risale l'albero oppure viene eseguita qualche rotazione ed il ciclo termina.

In realtà vi sono sei casi da considerare nel ciclo **while**, ma tre sono simmetrici agli altri tre a seconda che il padre di x, $p[x]$, sia il figlio sinistro oppure il figlio destro del nonno di x, $p[p[x]]$, che è individuato nella linea 4. Nella procedura è stato fornito solo il codice del caso in cui $p[x]$ risulti figlio sinistro. Si è anche fatta un'ipotesi di lavoro importante stabilendo che la radice sia sempre nera – proprietà che viene garantita dalla linea 18 ogni volta che il procedimento di inserzione termina – per cui $p[x]$ non è la radice e quindi $p[p[x]]$ esiste.

Il caso 1 si distingue dai casi 2 e 3 per il colore del fratello del padre di x, detto "zio". Nella linea 5 ad y è assegnato il puntatore $right[p[p[x]]]$ allo zio di x; nella linea 6 è eseguito un controllo per cui se y è rosso si esegue il caso 1 altrimenti il controllo passa ai casi 2 e 3. In ogni caso il nonno di x, $p[p[x]]$, è nero: infatti il padre $p[x]$ è rosso e la proprietà 3 è violata solo tra x e $p[x]$.

La situazione del caso 1 (linee 7-10) è mostrata nella figura 14.5. Il caso 1 è eseguito quando sia $p[x]$ che y sono rossi. Dato che $p[p[x]]$ è nero, si colorano $p[x]$ e y di nero e $p[p[x]]$ di rosso, risolvendo così il problema che x e $p[x]$ sono rossi adiacenti e mantenendo la proprietà 4, ma a questo punto potrebbe essere violata la proprietà 3 se il padre di $p[p[x]]$ è rosso: quindi si ripete il ciclo **while** assegnando ad x il valore $p[p[x]]$.

Figura 14.4 *L'esecuzione di* RB-INSERT. (a) *Un nodo x dopo l'inserzione. Dato che x e suo padre p[x] sono entrambi rossi, si verifica la violazione della proprietà 3. Dato che y, lo zio di x, è rosso, si può applicare il codice previsto per il caso 1. Si cambia il colore dei nodi ed il puntatore x è spostato nell'albero verso l'alto; l'albero ottenuto è mostrato in* (b). *Ancora una volta x e suo padre sono entrambi rossi, ma y, lo zio di x, è nero. Dato che x è il figlio destro di p[x], può essere applicato il caso 2. Viene eseguita una rotazione sinistra e l'albero risultante è mostrato in* (c). *A questo punto x è il figlio sinistro di suo padre per cui si applica il caso 3. Una rotazione destra produce l'albero in* (d), *che è un albero RB.*

Nei casi 2 e 3 il colore dello zio di x è nero. I due casi sono distinti in base al fatto che x sia un figlio destro oppure sinistro di $p[x]$. Le linee 12-13 rappresentano il caso 2, che insieme al caso 3 è mostrato nella figura 14.6. Nel caso 2, x è un figlio destro di suo padre, per cui si esegue subito una rotazione sinistra per trasformare questa situazione nel caso 3 (presentato nelle linee 14-16), in cui x è un figlio sinistro. Questa rotazione non influenza né la b-altezza dei nodi né la validità della proprietà 4, in quanto sia x che $p[x]$ sono rossi. Arrivando direttamente al caso 3 o passando dal caso 2 si verifica che y, lo zio di x, è un nodo nero dato che altrimenti si sarebbe eseguito il caso 1. A questo punto dobbiamo effettuare alcuni cambiamenti di colore dei nodi ed una rotazione destra che preserva la proprietà 4, dopo di

Figura 14.5 *Il caso 1 della procedura* RB-INSERT. *La proprietà 3 è violata dato che x e suo padre p[x] sono entrambi rossi. Si eseguono le stesse azioni sia che* (a) *x risulti figlio destro oppure che* (b) *x sia un figlio sinistro. Ciascuno dei sottoalberi α, β, γ, δ e ε ha una radice nera ed ha la stessa b-altezza. Il codice per il caso 1 cambia il colore di alcuni nodi, preservando la proprietà 4: tutti i cammini verso il basso da un certo nodo ad una radice hanno sempre lo stesso numero di nodi neri. Il ciclo* while *continua con il nonno di x, il nodo p[p[x]], come il nuovo x. A questo punto qualsiasi violazione della proprietà 3 può avvenire solo tra il nuovo x, che è rosso, e suo padre se anch'esso è rosso.*

che l'esecuzione termina in quanto non ci sono più due nodi rossi adiacenti. Il corpo del while non è ripetuto un'altra volta in quanto ormai $p[x]$ è nero.

Qual è il tempo di esecuzione della procedura RB-INSERT? Dato che l'altezza di un RB-albero di n nodi è $O(\lg n)$, la chiamata TREE-INSERT costa un tempo $O(\lg n)$. Il ciclo while è ripetuto solo se si esegue il caso 1 con il conseguente spostamento verso la radice del puntatore x. Per cui il numero totale di volte che il ciclo while può essere ripetuto è $O(\lg n)$. Quindi RB-INSERT impiega un tempo totale di $O(\lg n)$. Vale la pena osservare che non si eseguono più di due rotazioni in quanto se si esegue il caso 2 o il caso 3 il ciclo termina.

Figura 14.6 *I casi 2 e 3 della procedura* RB-INSERT. *Come nel caso 1, la proprietà 3 è violata sia nel caso 2 sia nel caso 3 in quanto x e suo padre p[x] sono entrambi rossi. Ciascuno dei sottoalberi α, β, γ e δ ha una radice nera e tutti hanno la stessa b-altezza. Il caso 2 è trasformato nel caso 3 attraverso una rotazione sinistra, che preserva la proprietà 4: tutti i cammini verso il basso da un certo nodo ad una radice hanno sempre lo stesso numero di nodi neri. Il caso 3 implica alcuni cambi di colore ed una rotazione destra, che preserva la proprietà 4. Dopo, il ciclo del* while *termina in quanto la proprietà 3 è ormai soddisfatta: sull'albero non ci sono altri due nodi rossi adiacenti.*

Esercizi

14.3-1 Nella linea 2 di RB-INSERT si assegna il colore rosso al nuovo nodo inserito. Si noti che assegnando ad x il colore nero la proprietà 3 non potrebbe essere violata. Perché allora non si è scelto il colore nero per x?

14.3-2 Nella linea 18 di RB-INSERT si assegna il colore nero alla radice dell'albero. Qual è il vantaggio?

14.3-3 Mostrare gli RB-alberi che risultano dalle inserzioni successive delle chiavi 41, 38, 31, 12, 19, 8 in un RB-albero inizialmente vuoto.

14.3-4 Si supponga che la b-altezza di ciascuno dei sottoalberi α, β, γ, δ, ε nelle figure 14.5 e 14.6 sia k. Etichettare ciascun nodo in ogni figura con la sua b-altezza per verificare che sia mantenuta la proprietà 4 dopo la trasformazione indicata.

14.3-5 Si consideri un RB-albero di n nodi costruito con inserimenti successivi tramite la procedura RB-INSERT. Mostrare che se $n > 1$ allora l'albero ha almeno un nodo rosso.

14.3-6 Descrivere come realizzare efficientemente RB-INSERT se la rappresentazione degli RB-alberi non prevede memoria per il puntatore al padre.

14.4 Cancellazione

Come per le altre operazioni di base su un RB-albero di n nodi, la cancellazione di un nodo impiega un tempo $O(\lg n)$. La cancellazione di un nodo da un RB-albero è, però, leggermente più complicata dell'inserimento di un nodo.

Per semplificare nel codice la gestione dei casi limite si usa una sentinella invece del valore NIL (si veda il paragrafo 11.2). Per un RB-albero T, $nil[T]$ è un oggetto con gli stessi campi di un comune nodo dell'albero. Il valore del suo campo *color* è BLACK mentre agli altri campi – *p*, *left*, *right* e *key* – sono assegnati valori arbitrari. Nel RB-albero tutti i puntatori a NIL sono sostituiti con puntatori alla sentinella $nil[T]$.

Si usano le sentinelle in modo da gestire un figlio NIL di un nodo x come un comune nodo il cui padre sia x. Si potrebbero aggiungere tanti nodi sentinella quanti sono i NIL nell'albero, così che il padre di ogni NIL sarebbe ben definito, ma ciò richiederebbe un certo spreco di memoria. Si usa, invece, una sola sentinella $nil[T]$ per rappresentare tutti i NIL. In questo caso quando si deve manipolare un nodo figlio di x si dovrà fare attenzione ad assegnare prima x a $p[nil[T]]$.

La procedura RB-DELETE è una semplice modifica della procedura TREE-DELETE (paragrafo 13.3). Dopo aver estratto un nodo, RB-DELETE richiama una procedura ausiliaria RB-DELETE-FIXUP che ha il compito di cambiare i colori ed eseguire le rotazioni necessarie al mantenimento delle proprietà RB.

```
RB-DELETE (T, z)
 1  if left[z] = nil[T] o right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          ▷ Si copiano anche i dati satellite di y.
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

Vi sono tre differenze tra le procedure TREE-DELETE e RB-DELETE. Per prima cosa, tutti i riferimenti a NIL in TREE-DELETE sono stati sostituiti con riferimenti alla sentinella *nil*[*T*] in RB-DELETE. In secondo luogo, il controllo per verificare se *x* è NIL nella linea 7 di TREE-DELETE è stato rimosso e l'assegnamento $p[x] \leftarrow p[y]$ è eseguito incondizionatamente nella linea 7 di RB-DELETE. Per cui, se *x* è la sentinella *nil*[*T*], il suo puntatore al padre si riferisce al padre del nodo *y* estratto. La terza differenza è la chiamata alla procedura RB-DELETE-FIXUP che è eseguita nella linea 17 se *y* è nero. Se *y* è rosso, le proprietà RB rimangono valide anche quando il nodo *y* è estratto, dato che in questo caso nessuna b-altezza è modificata nell'albero e non ci sono nodi rossi adiacenti. Il nodo *x* passato a RB-DELETE-FIXUP è il nodo che era l'unico figlio di *y* prima che *y* fosse estratto se *y* aveva un figlio diverso da NIL, oppure la sentinella *nil*[*T*] se *y* non aveva figli. Nell'ultimo caso, l'assegnamento incondizionato nella linea 7 assicura che il padre di *x* sia adesso il nodo che precedentemente era il padre di *y*, sia nel caso che *x* fosse un nodo interno con chiave, sia nel caso che fosse la sentinella *nil*[*T*].

Si può adesso esaminare come la procedura RB-DELETE-FIXUP ripristini le proprietà RB dell'albero di ricerca.

```
RB-DELETE FIXUP(T, x)
 1  while x ≠ root[T] e color[x] = BLACK
 2     do if x = left[p[x]]
 3           then w ← right[p[x]]
 4                if color[w] = RED
 5                   then color[w] ← BLACK                    ▷ caso 1
 6                        color[p[x]] ← RED                   ▷ caso 1
 7                        LEFT-ROTATE(T, p[x])                ▷ caso 1
 8                        w ← right[p[x]]                     ▷ caso 1
 9                if color[left[w] = BLACK e color[right[w]] = BLACK
10                   then color[w] ← RED                      ▷ caso 2
11                        x ← p[x]                            ▷ caso 2
12                   else if color[right[w]] = BLACK
13                           then color[left[w]] ← BLACK      ▷ caso 3
14                                color[w] ← RED              ▷ caso 3
15                                RIGHT-ROTATE(T, w)          ▷ caso 3
16                                w ← right[p[x]]             ▷ caso 3
17                        color[w] ← color[p[x]]              ▷ caso 4
18                        color[p[x]] ← BLACK                 ▷ caso 4
19                        color[right[w]] ← BLACK             ▷ caso 4
20                        LEFT-ROTATE(T, p[x])                ▷ caso 4
21                        x ← root[T]                         ▷ caso 4
22        else (analogo al ramo then con "right" e "left" scambiati)
23  color[x] ← BLACK
```


Se il nodo *y* estratto dalla procedura RB-DELETE è nero, allora la sua rimozione fa sì che qualsiasi cammino che contenesse *y* ha ora un nodo nero in meno. Quindi la proprietà 4 risulta violata da qualsiasi antenato di *y* nell'albero. Si può ovviare a questo problema considerando che il nodo *x* abbia un colore nero "extra"; cioè se si addiziona 1 al numero di nodi neri di qualsiasi cammino che contiene *x*, allora la proprietà 4 è soddisfatta. Quando si estrae un nodo *y* nero, si forza la sua stessa colorazione sul figlio *x*; il problema però è che facendo così un nodo *x* potrebbe essere "doppiamente colorato di nero", violando in questo modo la proprietà 1.

La procedura RB-DELETE-FIXUP ha il compito di ripristinare la proprietà 1. Lo scopo del ciclo **while** nelle linee 1-22 è di spostare il colore nero di troppo verso la radice dell'albero finché (1) *x* punta ad un nodo rosso, nel qual caso è colorato di nero nella linea 23; (2) *x* punta alla radice, nel qual caso la colorazione nera in più è semplicemente "rimossa"; oppure (3) possono essere eseguite rotazioni e ricolorazioni appropriate.

Dentro il ciclo **while**, *x* punta sempre ad un nodo nero, diverso dalla radice e con un colore nero extra. Nella linea 2 si verifica se *x* sia un figlio sinistro o destro di suo padre *p*[*x*]. (Abbiamo fornito il codice per il caso in cui *x* sia un figlio sinistro; il codice per il caso in cui *x* fosse figlio destro – come riportato nella linea 22 – è simmetrico.) Si mantiene un puntatore *w* al fratello di *x*: dato che il nodo *x* è doppiamente nero, il nodo *w* non può essere *nil*[*T*] perché altrimenti il numero di nodi neri nel cammino da *p*[*x*] alla foglia NIL che è *w* sarebbe più piccolo del numero nel cammino da *p*[*x*] a *x*.

I quattro casi segnalati nel codice sono illustrati nella figura 14.7. Prima di esaminarli in dettaglio, è conveniente capire più in generale come le trasformazioni eseguite preservino la proprietà 4 in ciascun caso. L'idea chiave è che, in ciascun caso, il numero dei nodi neri dalla radice (inclusa) del sottoalbero mostrato a ognuno dei sottoalberi $\alpha, \beta, \ldots, \zeta$ è mantenuto dalla trasformazione. Per esempio, nella figura 14.7(a), che illustra il caso 1, il numero di nodi neri dalla radice al sottoalbero α o al sottoalbero β è 3, sia prima che dopo la trasformazione (si ricordi che il puntatore *x* aggiunge un colore nero extra). Allo stesso modo, il numero di nodi

Figura 14.7 *I casi del ciclo* **while** *della procedura* RB-DELETE-FIXUP. *I nodi in neretto sono quelli neri, quelli grigi scuri sono i nodi rossi e quelli grigi chiari sono quelli che possono essere sia di colore nero che rosso e sono rappresentati da c e c'. Le lettere α, β, ..., ζ rappresentano sottoalberi arbitrari. In ciascun caso la configurazione di sinistra è trasformata nella configurazione di destra eseguendo qualche cambio di colore e/o eseguendo una rotazione. Un nodo, puntato da x, ha un colore nero in più. L'unico caso che implica la ripetizione del ciclo è il caso* 2. (a) *Il caso* 1 *è trasformato nel caso* 2, 3 *o* 4 *scambiando il colore tra i nodi B e D ed eseguendo una rotazione sinistra.* (b) *Nel caso* 2, *il colore nero in più rappresentato dal puntatore x è spostato nell'albero verso l'alto colorando il nodo D di rosso e facendo puntare ad x il nodo B. Se si arriva al caso* 2 *passando dal caso* 1, *il ciclo del* **while** *termina in quanto il colore c è rosso.* (c) *Il caso* 3 *è trasformato nel caso* 4 *scambiando i colori dei nodi C e D ed eseguendo una rotazione destra.* (d) *Nel caso* 4, *il colore nero in più rappresentato dal puntatore x può essere rimosso cambiando alcune colorazioni ed eseguendo una rotazione sinistra (senza violare alcuna delle proprietà degli alberi RB), quindi il ciclo termina.*

neri dalla radice a ciascuno dei sottoalberi γ, δ, ε e ζ è 2, sia prima che dopo la trasformazione. Nella figura 14.7(b), il calcolo deve includere il colore c, che può essere sia nero che rosso. Se si definisce count(RED) = 0 e count(BLACK) = 1, allora il numero di nodi neri dalla radice ad α è 2 + count(c), sia prima che dopo la trasformazione. In modo analogo, si possono verificare gli altri casi (Esercizio 14.4-5).

Il caso 1 (linee 5-8 della procedura RB-DELETE-FIXUP e figura 14.7(a)) si verifica quando il nodo w, fratello del nodo x, è rosso. Dato che w deve avere i figli neri, si possono scambiare

i colori di w e $p[x]$ e quindi eseguire una rotazione sinistra su $p[x]$ senza violare nessuna delle proprietà degli RB-alberi. Il nuovo fratello di x, uno dei figli di w, adesso è nero per cui si è trasformato il caso 1 in uno dei casi 2, 3 o 4.

I casi 2, 3 e 4 si manifestano quando il nodo w è nero: essi si distinguono in base al colore dei figli di w. Nel caso 2 (linee 10-11 della procedura RB-DELETE-FIXUP e figura 14.7(b)) entrambi i figli di w sono neri. Dato che anche w è nero, si toglie un colore nero sia a w che a x, lasciando x con un solo colore nero, w di colore rosso ed aggiungendo un colore nero a $p[x]$. Successivamente si ripete l'esecuzione del ciclo **while** con $p[x]$ come nuovo x. Si osservi che se si arriva al caso 2 passando dal caso 1, il colore c del nuovo nodo x è rosso, dato che il $p[x]$ originario era rosso; il ciclo quindi termina quando si torna a controllare la condizione del **while**.

Il caso 3 (linee 13-16 e figura 14.7(c)) si verifica quando w è nero, il suo figlio sinistro è rosso ed il suo figlio destro è nero. Si possono scambiare i colori tra w ed il suo figlio sinistro *left*[w] e poi eseguire una rotazione destra su w senza violare alcuna delle proprietà degli RB-alberi. Il nuovo fratello w di x è adesso un nodo nero con un figlio destro rosso per cui si è trasformato il caso 3 nel caso 4.

Il caso 4 (linee 17-21 e figura 14.7(d)) si verifica quando il fratello w di x è nero ed il figlio destro di w è rosso. Facendo alcuni scambi di colori ed eseguendo una rotazione sinistra su $p[x]$, si può eliminare il colore nero di troppo del nodo x senza che sia violata nessuna delle proprietà RB. L'assegnamento ad x del puntatore alla radice dell'albero serve per far terminare il ciclo **while** al successivo controllo della condizione del **while**.

Qual è il tempo di esecuzione di RB-DELETE? Dato che l'altezza di un RB-albero di n nodi è $O(\lg n)$, il costo totale della procedura, senza la chiamata a RB-DELETE-FIXUP, è di $O(\lg n)$. Dentro la procedura RB-DELETE-FIXUP i casi 1, 3 e 4 fanno terminare la procedura dopo l'esecuzione di un numero costante di cambiamenti di colorazione e al più tre rotazioni. Il caso 2 è l'unico per il quale si potrebbe ripetere l'esecuzione del ciclo **while**, spostando il puntatore x nell'albero verso l'alto per al più $O(\lg n)$ volte e senza rotazioni. Quindi la procedura RB-DELETE-FIXUP impiega un tempo $O(\lg n)$ ed esegue al più tre rotazioni, e anche il tempo di esecuzione complessivo della procedura RB-DELETE è $O(\lg n)$.

Esercizi

14.4-1 Mostrare che, se il colore della radice di un RB-albero è nero prima dell'esecuzione di RB-DELETE, allora è nero anche dopo.

14.4-2 Nell'Esercizio 14.3-3 si è trovato l'RB-albero che risulta dalle successive inserzioni delle chiavi 41, 38, 31, 12, 19, 8 in un albero inizialmente vuoto. Mostrare ora gli RB-alberi che risultano dalle cancellazioni successive delle chiavi nell'ordine 8, 12, 19, 31, 38, 41.

14.4-3 In quali linee della procedura RB-DELETE-FIXUP si potrebbe esaminare o modificare la sentinella *nil*[T]?

14.4-4 Semplificare il codice di LEFT-ROTATE usando una sentinella per NIL ed un'altra per mantenere il puntatore alla radice.

14.4-5 Per ciascuno dei casi della figura 14.7, fornire il numero di nodi neri dalla radice a ciascuno dei sottoalberi $\alpha, \beta, \ldots, \zeta$ e verificare che ciascun numero dopo la trasformazione rimane inalterato. Quando un nodo ha come colore c o c', usare nei calcoli la notazione count(c) o count(c').

14.4-6 Si supponga che un nodo x sia inserito con la procedura RB-INSERT e quindi sia subito cancellato con RB-DELETE. L'albero risultante è uguale a quello iniziale? Giustificare la risposta.

Problemi

14.1 Insiemi dinamici persistenti

Durante la vita di un algoritmo, talvolta si ha la necessità di memorizzare le versioni passate di un insieme dinamico che viene aggiornato. Un tale insieme è chiamato ***persistente***. Un modo per realizzare un insieme persistente consiste nel ricopiare tutto l'insieme ogni volta che è modificato, ma questo approccio può rallentare molto l'esecuzione del programma ed anche richiedere troppo spazio. Talvolta si può fare di meglio.

Si consideri un insieme persistente S con le operazioni INSERT, DELETE e SEARCH realizzato usando alberi binari di ricerca come mostrato nella figura 14.8(a). Si mantiene una radice separata per ogni versione dell'insieme. Per inserire la chiave 5 nell'insieme, si crea un nuovo nodo con chiave 5. Questo nodo diventa il figlio sinistro di un nuovo nodo con chiave 7, poiché il nodo con chiave 7 già esiste e non può essere modificato. Analogamente, il nuovo nodo con chiave 7 diventa il figlio sinistro di un nodo nuovo con chiave 8 il cui figlio destro è il nodo con chiave 10 che già esiste. Il nuovo nodo con chiave 8 diventa a sua volta il figlio destro di una nuova radice r' con chiave 4 il cui figlio sinistro è il nodo con chiave 3 già esistente. Quindi si ricopia solo una parte dell'albero, condividendo alcuni nodi che erano presenti nell'albero di partenza, come mostrato nella figura 14.8(b). Si assuma che ciascun nodo abbia i campi *key*, *left* e *right* ma non il campo per il padre. (Si veda anche l'Esercizio 14.3-6.)

a. Per un generico albero binario persistente di ricerca, identificare i nodi che devono essere cambiati per inserire una chiave k o cancellare un nodo y.

b. Scrivere una procedura PERSISTENT-TREE-INSERT che, dati un albero persistente T ed una chiave k da inserire, restituisce un nuovo albero persistente T' che rappresenta il risultato dell'inserzione di k in T.

c. Se l'altezza di un albero binario persistente di ricerca T è h, quali sono il tempo e lo spazio impiegati dalla procedura PERSISTENT-TREE-INSERT? (Lo spazio impiegato è proporzionale al numero di nuovi nodi inseriti.)

d. Si supponga di aver incluso in ciascun nodo anche il campo padre. In questo caso sarebbero necessarie ulteriori ricopiature da parte di PERSISTENT-TREE-INSERT. Provare che in questo caso PERSISTENT-TREE-INSERT richiederebbe un tempo ed uno spazio $\Omega(n)$, dove n è il numero dei nodi nell'albero.

e. Mostrare l'uso di RB-alberi per garantire che ogni inserzione o cancellazione richieda tempo e spazio pari a $O(\lg n)$, nel caso peggiore.

Figura 14.8 (a) *Un albero binario di ricerca con chiavi* 2, 3, 4, 7, 8, 10. (b) *L'albero binario persistente di ricerca che risulta dall'inserzione della chiave* 5. *La versione più recente dell'insieme è costituita dai nodi raggiungibili dalla radice r' e la versione precedente è composta dai nodi raggiungibili dalla radice r. I nodi grigi scuri sono aggiunti quando si inserisce la chiave* 5.

14-2 Operazione di giunzione (join) su RB-alberi

L'operazione di ***join*** prende due insiemi dinamici S_1 e S_2 ed un elemento x tale che per ogni $x_1 \in S_1$ e $x_2 \in S_2$, si abbia $key[x_1] \le key[x] \le key[x_2]$. Essa restituisce un insieme $S = S_1 \cup \{x\} \cup S_2$. In questo problema si esamina la realizzazione di questa operazione su RB-alberi.

a. Dato un RB-albero T, si memorizza la sua b-altezza nel campo $bh[T]$. Spiegare come questo campo possa essere mantenuto da RB-INSERT e RB-DELETE senza richiedere ulteriore memoria e senza aumentare il tempo di esecuzione asintotico. Mostrare che mentre si discende l'albero T, si può determinare la b-altezza di ciascun nodo in tempo $O(1)$ per ogni nodo visitato.

Si desidera realizzare l'operazione RB-JOIN(T_1, x, T_2) che distrugge T_1 e T_2 restituendo un RB-albero $T = T_1 \cup \{x\} \cup T_2$. Sia n il numero totale di nodi in T_1 e T_2.

b. Si assuma senza perdere di generalità che $bh[T_1] \ge bh[T_2]$. Descrivere un algoritmo con tempo di esecuzione $O(\lg n)$ che trovi in T_1, fra i nodi che hanno una b-altezza uguale a $bh[T_2]$, il nodo y nero con la chiave più grande.

c. Sia T_y il sottoalbero radicato in y. Descrivere come T_y possa essere sostituito da $T_y \cup \{x\} \cup T_2$ in un tempo $O(1)$ senza violare le proprietà di un albero binario di ricerca.

d. Qual è il colore da assegnare ad x perché le proprietà 1, 2 e 4 degli RB-alberi siano mantenute? Descrivere come possa essere fatta valere la proprietà 3 in un tempo $O(\lg n)$.

e. Spiegare perché il tempo di esecuzione di RB-JOIN è $O(\lg n)$.

Note al capitolo

L'idea del bilanciamento di un albero di ricerca è dovuta a Adel'son-Vel'skiĭ e Landis [2], che presentarono nel 1962 una classe di alberi di ricerca bilanciati poi chiamati "alberi AVL". Negli alberi AVL il bilanciamento è mantenuto attraverso le rotazioni, e possono essere richieste anche $\Theta(\lg n)$ rotazioni per mantenere il bilanciamento dopo una cancellazione in un albero con n nodi. Un'altra classe di alberi di ricerca, chiamati "alberi 2-3", fu presentata da J. E. Hopcroft

(non pubblicato) nel 1970. In un albero 2-3, il bilanciamento è mantenuto con la manipolazione dei gradi dei nodi dell'albero. Ad una generalizzazione degli alberi 2-3, introdotta da Bayer e McCreight [18] e chiamata "B-alberi", è dedicato il Capitolo 19.

Gli RB-alberi furono inventati da Bayer [17] con il nome di "B-alberi binari simmetrici". Guibas e Sedgewick [93] studiarono estesamente le loro proprietà ed introdussero la convenzione rosso/nero.

Vi sono molte altre varianti degli alberi binari bilanciati: forse i più affascinanti sono gli "alberi splay" presentati da Sleator e Tarjan [177], che sono "auto-aggiustanti". (Una descrizione dettagliata di questo tipo di alberi è data da Tarjan [188]). Gli alberi splay mantengono il bilanciamento senza alcuna esplicita condizione di bilanciamento come il colore. D'altra parte, le operazioni "splay" (che includono rotazioni) sono eseguite nell'albero tutte le volte che si fa un accesso all'albero. Il costo ammortizzato (si veda il Capitolo 18) di ciascuna operazione su un albero con n nodi è $O(\lg n)$.

Estensione di strutture di dati

15

Vi sono molti casi in cui la programmazione non richiede niente di più che strutture di dati da "manuale", come per esempio liste bidirezionali, tabelle hash o alberi binari, ma vi sono anche molti altri casi in cui si deve ricorrere a strutture di dati più originali. In queste circostanze non sarà sempre necessario progettare una nuova struttura di dati per intero: spesso sarà sufficiente *estendere* una struttura di dati classica aggiungendovi ulteriori informazioni e quindi definire su di essa le nuove operazioni che permetteranno di ottenere il comportamento richiesto dall'applicazione. L'estensione di una struttura di dati, tuttavia, non è sempre un'operazione semplice, dato che le informazioni aggiunte dovranno essere mantenute ed aggiornate in modo corretto dalle operazioni della corrispondente struttura di dati di base.

In questo capitolo si discuteranno due strutture di dati che sono costruite a partire dalla struttura a RB-albero. Il paragrafo 15.1 descrive una struttura di dati che può essere utilizzata per le operazioni di selezione su un insieme dinamico e mediante la quale si potrà determinare velocemente l'i-esimo elemento più piccolo o il rango di un dato elemento nell'ordinamento totale dell'insieme. Il paragrafo 15.2 generalizza il processo di estensione di una struttura di dati e presenta un teorema che potrà semplificare l'estensione di RB-alberi. Nel paragrafo 15.3 questo teorema è usato come supporto al progetto di una struttura di dati per la gestione di un insieme dinamico di intervalli, come per esempio intervalli temporali. Con una tale struttura di dati, dato un certo intervallo di tempo, sarà possibile determinare velocemente un intervallo dell'insieme che gli si sovrappone.

15.1 Selezione su un insieme dinamico

Nel Capitolo 10 si è introdotto il concetto di selezione su un insieme; in particolare l'i-esimo elemento di un insieme di n elementi, dove $i \in \{1, 2, \ldots, n\}$, è, semplicemente, l'elemento dell'insieme con la i-esima chiave più piccola. Si è verificato come la determinazione di un tale elemento su un insieme non ordinato costi tempo $O(n)$. In questo paragrafo si vedrà come un RB-albero può essere modificato per risolvere il problema della selezione in tempo $O(\lg n)$. Si vedrà anche come sia possibile nello stesso modo determinare il ***rango*** di un elemento – la sua posizione nell'insieme ordinato – in tempo $O(\lg n)$.

Nella figura 15.1 è mostrata una struttura di dati che consente operazioni di selezione efficienti. Un ***albero di selezione*** T è semplicemente un RB-albero con informazioni addizionali memorizzate in ogni nodo. In altre parole, oltre ai consueti campi di un RB-albero, cioè $key[x]$, $color[x]$, $p[x]$, $left[x]$ e $right[x]$ per un nodo x, si avrà anche il campo $size[x]$,

Questo campo contiene il numero dei nodi (interni) del sottoalbero radicato in x (x incluso), cioè la dimensione del sottoalbero. Definendo $size[\text{NIL}] = 0$, si ha la seguente identità:

$$size[x] = size[left[x]] + size[right[x]] + 1.$$

Per gestire correttamente i casi limite per il valore NIL, una realizzazione effettiva potrebbe verificare esplicitamente se il nodo argomento è NIL tutte le volte che si accede al campo *size* o più semplicemente, come già visto nel paragrafo 14.4, usare una sentinella $nil[T]$ per rappresentare NIL, ponendo $size[nil[T]] = 0$.

Ricerca di un elemento con un dato rango

Prima di mostrare come l'informazione sulla dimensione possa essere gestita durante le operazioni di inserzione e cancellazione, si esamineranno le realizzazioni di due algoritmi di selezione che utilizzano questa informazione aggiunta. Cominceremo analizzando un'operazione di ricerca di un elemento dato il suo rango. La procedura OS-SELECT(x, i) restituisce il puntatore al nodo che contiene la i-esima chiave più piccola nel sottoalbero radicato in x. (L'acronimo "OS" deriva da *Order-Statistics*, la denominazione inglese di questo tipo di operazioni.) Per trovare la i-esima chiave più piccola in un albero di selezione T si richiamerà la procedura con OS-SELECT($root[T]$, i).

```
OS-SELECT(x, i)
1  r ← size[left[x]] + 1
2  if i = r
3     then return x
4     else if i < r
5             then return OS-SELECT(left[x], i)
6             else return OS-SELECT(right[x], i − r)
```

L'idea chiave della procedura OS-SELECT è simile a quella degli algoritmi di selezione visti nel Capitolo 10; il valore di $size[left[x]]$ è il numero di nodi che precedono x nella visita simmetrica del sottoalbero radicato in x. Quindi, $size[left[x]] + 1$ rappresenta il rango di x all'interno del sottoalbero radicato in x.

Nella linea 1 di OS-SELECT si calcola r, il rango di x all'interno del sottoalbero radicato in x; se $i = r$ allora il nodo x è proprio l'i-esimo elemento più piccolo per cui nella linea 3 si restituisce x. Se $i < r$ allora l'i-esimo elemento più piccolo è nel sottoalbero sinistro di x ed ecco perciò la chiamata ricorsiva della linea 5; se invece $i > r$, allora l'i-esimo elemento più piccolo è nel sottoalbero destro. Dato che vi sono r elementi nel sottoalbero radicato in x che precedono il sottoalbero destro di x nella visita simmetrica, allora l'i-esimo elemento più piccolo del sottoalbero radicato in x corrisponde all'$(i - r)$-esimo elemento più piccolo del sottoalbero radicato in $right[x]$. Questo elemento è determinato in modo ricorsivo nella linea 6 della procedura.

Per verificare come operi la procedura OS-SELECT si consideri la ricerca del diciassettesimo elemento più piccolo nell'albero di selezione mostrato nella figura 15.1. Si comincia con x come radice, la cui chiave è 26, e con $i = 17$. Dato che la dimensione del sottoalbero sinistro della radice è 12, il suo rango è 13, per cui il nodo con rango 17 è il quarto ($17 - 13 = 4$) elemento più piccolo del sottoalbero destro. Dopo la chiamata ricorsiva, x è il nodo con chiave

Figura 15.1 *Un albero di selezione, che è un albero RB esteso. I nodi grigi sono i nodi rossi mentre i nodi scuri sono quelli neri. Oltre agli usuali campi, ciascun nodo x ha un campo $size[x]$ che rappresenta il numero di nodi presenti nel sottoalbero radicato in x.*

41 e $i = 4$. Dato che la dimensione del sottoalbero sinistro del nodo con chiave 41 è 5, il suo rango nel sottoalbero è 6, per cui il nodo con rango 4 è il quarto elemento più piccolo del sottoalbero sinistro del nodo 41. Dopo la chiamata ricorsiva, x è il nodo con chiave 30 ed il suo rango nel suo sottoalbero è 2. Quindi si effettua una nuova chiamata ricorsiva, per trovare il secondo ($4 - 2 = 2$) elemento più piccolo, nel sottoalbero radicato nel nodo con chiave 38. Si trova ora che il suo sottoalbero sinistro ha dimensione 1 e ciò significa che tale nodo ha rango 2 ed è perciò il secondo elemento più piccolo. Quindi la procedura restituisce il puntatore al nodo con chiave 38.

Il tempo complessivo della procedura OS-SELECT è, al più, proporzionale all'altezza dell'albero, dato che ad ogni chiamata ricorsiva si scende di un livello. Dato che l'altezza di un RB-albero (e quindi anche di un albero di selezione) è $O(\lg n)$, dove n è il numero dei nodi, si ha che il tempo di esecuzione di OS-SELECT su un insieme dinamico di n elementi è $O(\lg n)$.

Determinazione del rango di un elemento

Dato il puntatore ad un nodo x di un albero di selezione T, la procedura OS-RANK restituisce la posizione di x nell'ordinamento lineare determinato con una visita simmetrica di T.

```
OS-RANK(T, x)
1  r ← size[left[x]] + 1
2  y ← x
3  while y ≠ root[T]
4     do if y = right[p[y]]
5           then r ← r + size[left[p[y]]] + 1
6        y ← p[y]
7  return r
```

La procedura opera come segue. Il rango di x può essere visto come il numero di nodi che precedono x nella visita simmetrica, più 1 per il nodo stesso. È sempre valida la seguente condizione invariante: all'inizio del ciclo **while** delle linee 3-6, r è il rango di $key[x]$ nel sottoalbero radicato in y. La condizione invariante è mantenuta nel seguente modo. Nella linea 1 si assegna ad r il rango di $key[x]$ nel sottoalbero radicato in x. L'assegnamento $y \leftarrow x$ rende vera la condizione invariante quando il test nella linea 3 è eseguito la prima volta. In ciascuna

iterazione del ciclo **while** si considera il sottoalbero radicato in $p[y]$. Si è già contato il numero di nodi che precedono x nella visita simmetrica del sottoalbero radicato in y, per cui si deve aggiungere: il numero di nodi nel sottoalbero radicato nel fratello di y che precedono x nella visita simmetrica, più 1 per $p[y]$ se $p[y]$ precede anch'esso x. Se y è un figlio sinistro, allora né $p[y]$ né qualsiasi nodo del sottoalbero destro di $p[y]$ precedono x per cui r rimane invariato. Altrimenti, y è un figlio destro e tutti i nodi del sottoalbero sinistro di $p[y]$ precedono x, compreso $p[y]$ stesso. Perciò nella linea 5 si aggiunge $size[left[p[y]]] + 1$ al valore di r. L'assegnamento $y \leftarrow p[y]$ mantiene l'invariante per l'iterazione successiva. Quando $y = root[T]$, allora la procedura restituisce il valore di r, che corrisponde proprio al rango di $key[x]$.

Si supponga, per esempio, di eseguire la procedura OS-RANK per trovare il rango del nodo con chiave 38 nell'albero di figura 15.1; all'inizio del ciclo **while** si ottiene la seguente sequenza di valori per $key[y]$ ed r:

iterazione	$key[y]$	r
1	38	2
2	30	4
3	41	4
4	26	17

Quindi la procedura restituisce il valore 17 come rango del nodo con chiave 38.

Dato che ciascuna iterazione del ciclo **while** impiega tempo $O(1)$ ed y sale nell'albero di un livello ad ogni iterazione, il tempo di esecuzione di OS-RANK è nel caso peggiore proporzionale all'altezza dell'albero: $O(\lg n)$ in un albero di selezione di n nodi.

Mantenimento della dimensione dei sottoalberi

Dato il campo *size* per ciascun nodo, le procedure OS-SELECT e OS-RANK possono calcolare velocemente informazioni utili per il problema della selezione. Ma tutto questo lavoro sarebbe stato inutile se questi campi non potessero essere mantenuti efficientemente dalle operazioni di base che modificano un RB-albero. Si vedrà adesso come le informazioni sulla dimensione dei sottoalberi possano essere mantenute sia nell'operazione di inserzione che di cancellazione di un nodo senza modificare i tempi di esecuzione asintotici di queste operazioni.

Nel paragrafo 14.3 si è osservato come l'operazione di inserzione in un RB-albero sia composta di due fasi. Nella prima fase si discende l'albero dalla radice inserendo il nuovo nodo come figlio di un nodo già esistente. Nella seconda fase si risale l'albero cambiando i colori ed eseguendo rotazioni in modo da rispettare le proprietà RB.

Per gestire la dimensione dei sottoalberi, nella prima fase è sufficiente incrementare $size[x]$ di 1 in ogni nodo x attraversato nel cammino dalla radice verso le foglie. Nel nodo aggiunto si pone size uguale a 1. Dato che sul cammino percorso ci sono $O(\lg n)$ nodi, il costo aggiuntivo per il mantenimento del campo *size* è $O(\lg n)$.

Nella seconda fase le modifiche alla struttura dell'albero sono causate solo dalle rotazioni che sono, al più, due. Inoltre la rotazione è un'operazione locale: modifica solo il campo *size* dei nodi incidenti sull'arco sul quale avviene la rotazione. Rispetto al codice della procedura LEFT-ROTATE(T, x) presentata nel paragrafo 14.2, sarà sufficiente aggiungere le seguenti linee:

```
13    size[y] ← size[x]
14    size[x] ← size[left[x]] + size[right[x]] + 1
```

Figura 15.2 *L'aggiornamento della dimensione dei sottoalberi durante le rotazioni. I due campi size che devono essere aggiornati sono solo quelli incidenti sull'arco su cui eseguire la rotazione. Gli aggiornamenti sono locali, dato che richiedono la conoscenza del campo size di x, y e delle radici dei sottoalberi mostrati come triangoli.*

La figura 15.2 mostra il modo in cui i campi sono aggiornati. La modifica della procedura RIGHT-ROTATE è simmetrica.

Dato che in un RB-albero sono operate, al più, due rotazioni durante la seconda fase dell'operazione di inserzione, il costo addizionale per aggiornare i campi *size* è $O(1)$. Quindi, il tempo complessivo per l'inserimento di un nodo in un albero di selezione di n nodi è $O(\lg n)$, asintoticamente uguale a quello per l'inserzione in un comune RB-albero.

Anche la cancellazione da un RB-albero è composta di due fasi: la prima opera sulla struttura di base dell'albero di ricerca, la seconda causa al più tre rotazioni oppure non prevede alcuna modifica strutturale (si veda il paragrafo 14.4). Nella prima fase si estrae un nodo y. Per aggiornare la dimensione dei sottoalberi è sufficiente risalire il cammino dal nodo y alla radice decrementando di 1 il valore del campo *size* di ciascun nodo sul cammino. Dato che in un RB-albero con n nodi, questo cammino ha lunghezza $O(\lg n)$, il tempo impiegato nella prima fase per il mantenimento del campo *size* è $O(\lg n)$. Le $O(1)$ rotazioni della seconda fase possono essere gestite nello stesso modo visto per l'inserzione. Quindi sia l'inserzione che la cancellazione, compresi i tempi per la gestione dei campi *size*, impiegano tempo $O(\lg n)$ per un albero di selezione composto da n nodi.

Esercizi

15.1-1 Mostrare come opera OS-Select(T, 10) sull'RB-albero T di figura 15.1.

15.1-2 Mostrare come opera OS-RANK(T, x) sull'RB-albero T di figura 15.1 con nodo x tale che $key[x] = 35$.

15.1-3 Scrivere una versione iterativa di OS-SELECT.

15.1-4 Scrivere una procedura ricorsiva OS-KEY-RANK(T, k) che prende come input un albero di selezione T e una chiave k e restituisce il rango di k nell'insieme dinamico rappresentato da T. Si assuma che le chiavi in T siano tutte diverse.

15.1-5 Dati un elemento x di un albero di selezione di n nodi ed un numero naturale i, come può essere determinato in tempo $O(\lg n)$ l'i-esimo successore di x secondo l'ordinamento degli elementi dell'albero?

15.1-6 Si osservi che il campo *size* è sempre utilizzato da OS-SELECT o da OS-RANK solo per calcolare il rango di x nel sottoalbero radicato in x. Di conseguenza, si supponga di memorizzare direttamente in ogni nodo il suo rango rispetto al sottoalbero di cui esso è la radice. Mostrare come questa informazione possa essere mantenuta nelle operazioni di inserzione e cancellazione. (Si ricordi che entrambe le operazioni possono richiedere rotazioni.)

15.1-7 Mostrare come si possa usare un albero di selezione per contare, in tempo $O(n \lg n)$, il numero di inversioni (si veda il Problema 1-3) in un array di dimensione n.

* *15.1-8* Si considerino n corde di un cerchio, ciascuna definita attraverso i suoi estremi. Descrivere un algoritmo con tempo $O(n \lg n)$ per la determinazione del numero di coppie di corde che si intersecano all'interno del cerchio. (Per esempio, se le n corde sono tutte diametri che si incontrano nel centro del cerchio, allora la soluzione corretta è $\binom{n}{2}$.) Si assuma che nessuna coppia di corde si tocchi in un estremo.

15.2 Come estendere una struttura di dati

Il processo di estensione di una struttura di dati di base in modo da offrire funzionalità addizionali si verifica abbastanza frequentemente nel progetto di algoritmi. Nel prossimo paragrafo si utilizzerà una struttura di dati opportunamente modificata per offrire operazioni su intervalli. In questo paragrafo si esamineranno i passi necessari per questa estensione della struttura di dati. Inoltre, sarà anche dimostrato un teorema che permette in molti casi di estendere facilmente gli RB-alberi.

L'estensione di una struttura di dati può essere suddivisa in quattro passi:

1. scelta della struttura di dati di base;
2. determinazione dell'informazione aggiunta che deve essere mantenuta nella struttura di dati di partenza;
3. verifica che l'informazione aggiunta possa essere mantenuta attraverso le usuali operazioni di modifica della struttura di dati di partenza;
4. sviluppo delle nuove operazioni.

Come prescrive qualsiasi metodo di progetto, non si deve procedere nello sviluppo dei diversi passi seguendo ciecamente un ordine prestabilito, anzi la maggior parte del progetto consiste di prove ed errori per cui lo sviluppo dei diversi passi, generalmente, procede in parallelo. Per esempio, la determinazione dell'informazione da aggiungere e lo sviluppo delle nuove operazioni (i passi 2 e 4) risultano interconnessi in quanto le nuove operazioni devono essere in grado di mantenere in modo efficiente le informazioni aggiunte. Ciò nonostante, questo metodo in quattro passi fornisce una guida che permette di focalizzare gli sforzi nell'estendere la struttura ed è anche un buon modo per organizzare la documentazione sulla nuova struttura di dati estesa.

Questi passi sono stati seguiti nel paragrafo 15.1 per la progettazione di alberi di selezione. Per il passo 1 si è scelto l'RB-albero come struttura di dati di base. Tale scelta è giustificata dal fatto che gli RB-alberi risultano efficienti quando sono impiegati per altre operazioni su insiemi dinamici che prevedono un ordinamento totale degli elementi dell'insieme, come MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR.

Per il passo 2 si è scelto il campo *size*, che in ogni nodo x memorizza la dimensione del sottoalbero radicato in x. Di solito, l'informazione aggiunta rende le operazioni più efficienti. Per esempio, le operazioni OS-SELECT e OS-RANK potevano essere realizzate usando solo le chiavi già memorizzate nell'albero, ma non sarebbero state eseguite con tempo $O(\lg n)$. Talvolta, l'informazione aggiunta è costituita da un puntatore piuttosto che da dati, come nell'Esercizio 15.2-1.

Per il passo 3, si è verificato che le operazioni di inserzione e cancellazione possano mantenere l'informazione del campo *size*, mentre il loro tempo di esecuzione $O(\lg n)$ rimane inalterato. In linea di principio, per mantenere l'informazione aggiunta, dovrebbe essere sufficiente un piccolo numero di modifiche alla struttura di dati. Per esempio, memorizzando direttamente in ciascun nodo il suo rango nell'albero, le operazioni OS-SELECT e OS-RANK risultano più veloci, ma l'inserimento di un nuovo elemento minimo implica la modifica di questa informazione in ogni nodo dell'albero. Invece quando si memorizza la dimensione del sottoalbero, l'inserimento di un nuovo nodo implica la modifica di questa informazione soltanto in $O(\lg n)$ nodi.

Per il passo 4, si sono sviluppate le operazioni OS-SELECT e OS-RANK. D'altra parte è proprio la necessità di realizzare nuove funzionalità che spinge all'estensione di strutture di dati. Più raramente si estende una struttura di dati, piuttosto che per sviluppare nuove operazioni, per rendere più efficienti le operazioni già previste, come nell'Esercizio 15.2-1.

Estensione di RB-alberi

Quando si utilizza come struttura di dati estesa un RB-albero, si può dimostrare che certi tipi di informazioni addizionali possono essere efficientemente mantenuti dalle operazioni di inserzione e cancellazione, in modo da rendere il passo 3 molto semplice. La dimostrazione del seguente teorema generalizza le argomentazioni del paragrafo 15.1 in cui si è esaminata la possibilità di aggiungere il campo *size* per trasformare un RB-albero in un albero di selezione.

Teorema 15.1 (Estensione di un RB-albero)

Sia f un campo che estende un RB-albero T composto da n nodi e si supponga che il valore di f per un nodo x possa essere calcolato usando soltanto le informazioni dei nodi: x, $left[x]$ e $right[x]$, compresi i valori di $f[left[x]]$ e $f[right[x]]$. Allora, durante l'inserzione e la cancellazione, si può mantenere l'informazione f su tutti i nodi di T senza influenzare asintoticamente il tempo $O(\lg n)$ di queste operazioni.

Dimostrazione. L'idea di base della dimostrazione è che una modifica del campo f di un nodo x si propaga nell'albero solo agli antenati di x. Cioè, la modifica di $f[x]$ può richiedere che $f[p[x]]$ sia aggiornato, ma niente di più; la modifica di $f[p[x]]$ può richiedere che $f[p[p[x]]]$ sia aggiornato, ma niente di più e così via fino alla cima all'albero. Dopo che $f[root[T]]$ è aggiornato, nessun altro nodo dipende dal nuovo valore, per cui il processo termina. Quindi, visto che l'altezza di un RB-albero è $O(\lg n)$, la modifica di un campo f di un nodo costa tempo $O(\lg n)$ per l'aggiornamento dei nodi che dipendono da tale modifica.

L'inserimento di un nodo x in un albero T è composto da due fasi (si veda il paragrafo 14.3). Nella prima fase x è inserito come figlio di un nodo $p[x]$ che già esisteva. Il valore di $f[x]$ può essere calcolato in tempo $O(1)$ dato che, per ipotesi, esso dipende solo dal contenuto degli altri campi di x stesso e dalle informazioni associate ai figli di x, che però sono entrambi NIL. Dopo che $f[x]$ è stato calcolato, la modifica è propagata all'indietro sull'albero, per cui il tempo complessivo impiegato nella prima fase dell'inserzione è $O(\lg n)$. Nella seconda fase, l'unica modifica strutturale all'albero deriva dalle rotazioni. Dato che in una rotazione cambiano solo due nodi, il tempo impiegato in ogni rotazione per l'aggiornamento dei campi f è $O(\lg n)$. Quindi, visto che al più ci sono due rotazioni per ogni inserzione, il tempo complessivo per l'inserzione è di $O(\lg n)$.

Come l'inserzione, anche la cancellazione è composta di due fasi (si veda il paragrafo 14.4). Nella prima fase si verifica una modifica all'albero solo se il nodo cancellato è sostituito dal suo successore e poi di nuovo quando o il nodo cancellato o il suo successore sono effettivamente estratti. La propagazione degli aggiornamenti di f causata da queste modifiche costa al più $O(\lg n)$ dato che gli aggiornamenti modificano l'albero localmente. Modificato l'RB-albero, la seconda fase richiede al più tre rotazioni e per ogni rotazione si impiega al più tempo $O(\lg n)$ per propagare gli aggiornamenti di f. Quindi, come per l'inserzione, il tempo complessivo per la cancellazione è $O(\lg n)$. ■

In molti casi, come per il mantenimento del campo *size* per gli alberi di selezione, il costo dell'aggiornamento per una rotazione è di $O(1)$, anziché $O(\lg n)$, come dimostrato nel Teorema 15.1. L'Esercizio 15.2-4 ne fornisce un esempio.

Esercizi

15.2-1 Mostrare come le operazioni di interrogazione su un insieme dinamico MINIMUM, MAXIMUM, SUCCESSOR e PREDECESSOR possano essere eseguite, su un albero di selezione esteso, in tempo $O(1)$ nel caso peggiore. Le prestazioni asintotiche delle altre usuali operazioni sull'albero di selezione non dovrebbero essere modificate.

15.2-2 È possibile mantenere la b-altezza dei nodi di un RB-albero come nuovo campo dei nodi senza per questo influire sulle prestazioni asintotiche di una qualsiasi operazione ordinaria sugli RB-alberi? Giustificare la risposta.

15.2-3 È possibile mantenere efficientemente la profondità dei nodi in un RB-albero come ulteriore campo dei nodi dell'albero? Giustificare la risposta.

* *15.2-4* Sia $\otimes$ un operatore binario associativo e sia a un campo mantenuto in ciascun nodo di un RB-albero. Si supponga di voler includere in ogni nodo x un ulteriore campo f tale che $f[x] = a[x_1] \otimes a[x_2] \otimes \cdots \otimes a[x_m]$, dove $x_1, x_2, \ldots, x_m$ è l'elenco in ordine simmetrico dei nodi del sottoalbero radicato in x. Mostrare che il campo f può essere aggiornato con un tempo $O(1)$ dopo una rotazione. Adattare il ragionamento seguito per mostrare che anche il campo *size* negli alberi di selezione può essere mantenuto con un tempo $O(1)$ per ogni rotazione.

* *15.2-5* Si desidera estendere gli RB-alberi con una nuova operazione RB-ENUMERATE(x, a, b) che restituisce tutte le chiavi k di un RB-albero radicato in x tali che $a \le k \le b$. Descrivere come RB-ENUMERATE possa essere realizzata con un algoritmo che impiega tempo $\Theta(m + \lg n)$, dove m è il numero di chiavi che sono restituite e n è il numero dei nodi interni dell'albero. (*Suggerimento*: non è necessario aggiungere nuovi campi ai nodi dell'RB-albero.)

15.3 Alberi di intervalli

In questo paragrafo gli RB-alberi saranno estesi per offrire operazioni su insiemi dinamici di intervalli. Un ***intervallo chiuso*** è una coppia ordinata di numeri reali $[t_1, t_2]$ dove $t_1 \le t_2$. L'intervallo $[t_1, t_2]$ rappresenta l'insieme $\{t \in \mathbb{R} : t_1 \le t \le t_2\}$. Intervalli ***aperti*** o ***semiaperti*** escludono dall'insieme rispettivamente entrambi o uno dei due estremi. In questo paragrafo si assumerà che gli intervalli siano chiusi; l'applicazione dei risultati ottenuti agli intervalli aperti e semiaperti è immediata.

Gli intervalli sono adatti alla rappresentazione di eventi ciascuno dei quali si manifesta in un periodo continuo di tempo. Per esempio, si potrebbe richiedere ad una base di dati di intervalli temporali, quali eventi si sono verificati durante un certo intervallo. La struttura di dati di questo paragrafo ben si presta ad essere utilizzata per rappresentare una tale base di dati di intervalli.

Si può rappresentare un intervallo $[t_1, t_2]$ come un oggetto i con campi $low[i] = t_1$ (l'***estremo sinistro***) e $high[i] = t_2$ (l'***estremo destro***). Si dice che due intervalli i ed i' si ***sovrappongono*** se $i \cap i' \ne \emptyset$, cioè, se $low[i] \le high[i']$ e $low[i'] \le high[i]$. Qualsiasi coppia di intervalli i ed i' soddisfa la proprietà di ***tricotomia degli intervalli***: cioè vale esattamente una delle tre seguenti proprietà:

a. i ed i' si sovrappongono,

b. $high[i] < low[i']$,

c. $high[i'] < low[i]$.

La figura 15.3 mostra queste tre possibilità.

Figura 15.3 *La tricotomia sugli intervalli per due intervalli chiusi i e i'.* (a) *Se i e i' si sovrappongono, allora ci sono quattro situazioni possibili: in ciascuna, $low[i] \le high[i']$ e $low[i'] \le high[i]$.* (b) $high[i] < low[i']$. (c) $high[i'] < low[i]$.

Figura 15.4 *Un albero di intervalli.* (a) *Un insieme di* 10 *intervalli, disegnato ordinato dal basso verso l'alto rispetto agli estremi sinistri.* (b) *L'albero di intervalli che li rappresenta. Una visita in ordine simmetrico elenca i nodi ordinati rispetto agli estremi sinistri.*

Un ***albero di intervalli*** è un RB-albero che memorizza un insieme dinamico di elementi, con ciascun elemento x contenente un intervallo $int[x]$. Sugli alberi di intervalli sono previste le seguenti operazioni.

INTERVAL-INSERT(T, x) aggiunge l'elemento x, il cui campo *int* si suppone contenga un intervallo, all'albero di intervalli T.

INTERVAL-DELETE(T, x) rimuove l'elemento x dall'albero di intervalli T.

INTERVAL-SEARCH(T, i) restituisce il puntatore ad un elemento x nell'albero di intervalli T tale che $int[x]$ si sovrappone all'intervallo i, oppure NIL se un tale elemento non è nell'insieme.

La figura 15.4 mostra come un albero di intervalli rappresenti un insieme di intervalli. Per esaminare la progettazione di un albero di intervalli e delle operazioni eseguite su di esso, si seguiranno i quattro passi del metodo proposto nel paragrafo 15.2.

Passo 1: struttura di dati di base

Si sceglie un RB-albero in cui ciascun nodo x contiene un intervallo $int[x]$ e la chiave di x corrisponde all'estremo sinistro, $low[int[x]]$, dell'intervallo. Quindi la visita in ordine simmetrico dell'albero elenca gli intervalli ordinati rispetto all'estremo sinistro.

Passo 2: informazione aggiunta

Oltre agli intervalli stessi, ciascun nodo x contiene un valore, $max[x]$, che è il più grande degli estremi degli intervalli memorizzati nel sottoalbero radicato in x. Dato che in qualsiasi intervallo l'estremo destro vale almeno quanto l'estremo sinistro, $max[x]$ contiene il più grande tra tutti gli estremi destri nel sottoalbero radicato in x.

Passo 3: mantenimento dell'informazione aggiunta

Si deve verificare che le operazioni di inserzione e di cancellazione possano essere eseguite in tempo $O(\lg n)$ su un albero di intervalli di n nodi. A tale scopo, se sono dati l'intervallo $int[x]$ ed i valori *max* dei nodi figli di x, si può determinare il valore di $max[x]$ nel modo seguente:

$$max[x] = \max(high[int[x]], max[left[x]], max[right[x]]).$$

Da questa formula, per il Teorema 15.1, si deduce che l'inserzione e la cancellazione possono essere eseguite in tempo $O(\lg n)$. Infatti, l'aggiornamento dei campi *max* dopo ogni rotazione può essere effettuato in tempo $O(1)$, come mostrato negli Esercizi 15.2-4 e 15.3-1.

Passo 4: sviluppo delle nuove operazioni

L'unica nuova operazione di cui si ha bisogno è INTERVAL-SEARCH(T, i), che trova nell'albero T un intervallo che si sovrappone all'intervallo i oppure restituisce NIL se non ci sono intervalli nell'albero che si sovrappongono a i.

```
INTERVAL-SEARCH(T, i)
x ← root[T]
while x ≠ NIL ed i non si sovrappone a int[x]
    do if  left[x] ≠ NIL e max[left[x]] ≥ low[i]
           then x ← left[x]
           else x ← right[x]
return x
```

La ricerca di un intervallo che si sovrappone a i comincia assegnando ad x il valore della radice dell'albero e procede quindi verso le foglie. Essa termina quando viene trovato un intervallo oppure quando x assume il valore NIL. Dato che ciascuna iterazione del ciclo impiega tempo $O(1)$ e dato che l'altezza di un RB-albero di n nodi è $O(\lg n)$, si conclude che la procedura INTERVAL-SEARCH impiega tempo $O(\lg n)$.

Prima di prendere in considerazione la correttezza della procedura INTERVAL-SEARCH, se ne esamini il funzionamento quando è applicata all'albero di intervalli della figura 15.4. Si supponga di voler trovare un intervallo che si sovrappone all'intervallo $i = [22, 25]$. Si comincia con x come radice, che contiene [16, 21] e quindi non si sovrappone a i. Dato che $max[left[x]] = 23$ è maggiore di $low[i] = 22$, il ciclo continua assegnando ad x il figlio sinistro della radice, il nodo contenente [8, 9], che a sua volta non si sovrappone a i. A questo punto,

$max[left[x]] = 10$ è più piccolo di $low[i] = 22$, per cui il ciclo continua con il figlio destro di x come nuovo x. L'intervallo [15, 23] memorizzato nel nodo si sovrappone a i, per cui la procedura restituisce il puntatore a questo nodo.

Come esempio di ricerca fallita, si supponga di voler trovare un intervallo che si sovrapponga a $i = [11, 14]$ nell'albero di intervalli della figura 15.4. Di nuovo si comincia con la radice come valore di x. Dato che l'intervallo della radice, [16, 21], non si sovrappone a i e dato che $max[left[x]] = 23$ è maggiore di $low[i] = 11$, si prosegue sul nodo sinistro che contiene l'intervallo [8, 9]. (Si noti che nessun intervallo nel sottoalbero destro si sovrappone a i, come si vedrà nel seguito.) L'intervallo [8, 9] non si sovrappone a i e $max[left[x]] = 10$ è più piccolo di $low[i] = 11$, per cui si va a destra. (Si noti che nessun intervallo nel sottoalbero sinistro si sovrappone a i.) L'intervallo [15, 23] non si sovrappone a i ed il figlio sinistro è NIL per cui si va a destra; quindi il ciclo termina e la procedura restituisce il valore NIL.

Per verificare la correttezza della procedura INTERVAL-SEARCH, si deve capire perché sia sufficiente esaminare un solo cammino a partire dalla radice. L'idea di base è che, in qualsiasi nodo x, se $int[x]$ non si sovrappone a i, la ricerca procede sempre in una direzione certa: se nell'albero c'è un intervallo che si sovrappone a i, prima o poi sarà trovato. Il seguente teorema enuncia questa proprietà in modo più preciso.

Teorema 15.2

Si consideri una qualsiasi iterazione del ciclo **while** della procedura INTERVAL-SEARCH(T, i).

1. Se è eseguita la linea 4 (la ricerca prosegue a sinistra), allora o il sottoalbero sinistro di x contiene un intervallo che si sovrappone a i oppure nessun intervallo nel sottoalbero destro di x si sovrappone a i.
2. Se è eseguita la linea 5 (la ricerca prosegue a destra), allora il sottoalbero sinistro di x non contiene alcun intervallo che si sovrapponga ad i.

Dimostrazione. La dimostrazione di entrambi i casi dipende dalla tricotomia degli intervalli. Si prova prima il caso 2 che è più semplice. Si osservi che se viene eseguita la linea 5, allora a causa della condizione della linea 3 si ha $left[x] = \text{NIL}$ oppure $max[left[x]] < low[i]$. Se $left[x] = \text{NIL}$ allora il sottoalbero radicato in $left[x]$ chiaramente non può contenere un intervallo che si sovrapponga a i, dato che non contiene alcun intervallo. Si supponga perciò che $left[x] \neq \text{NIL}$ e $max[left[x]] < low[i]$. Sia i' un intervallo del sottoalbero sinistro di x (si veda la figura 15.5(a)). Dato che $max[left[x]]$ è il più grande estremo destro degli intervalli del sottoalbero sinistro di x, allora si ha

$$\begin{aligned} high[i'] &\leq max[left[x]] \\ &< low[i], \end{aligned}$$

e quindi, per la tricotomia degli intervalli, i' e i non si sovrappongono, il che completa la dimostrazione del caso 2.

Per dimostrare il caso 1, si fa l'ipotesi che nel sottoalbero sinistro di x non esistano intervalli che si sovrappongano a i (dato che se ci fosse un intervallo la dimostrazione sarebbe già terminata). Sotto questa ipotesi, si deve solo dimostrare che nessun intervallo del sottoalbero destro di x possa sovrapporsi a i. Si osservi che se viene eseguita la linea 4, allora per la

Figura 15.5 *Gli intervalli nella dimostrazione del Teorema 15.2. Il valore di* $max[left[x]]$ *è mostrato in ciascun caso con una linea tratteggiata.* (a) *Caso 2: la ricerca continua a destra. Nessun intervallo* i' *può sovrapporsi a* i. (b) *Caso 1: la ricerca continua a sinistra. Il sottoalbero sinistro di* x *contiene un intervallo che si sovrappone a* i *(situazione non mostrata), oppure c'è un intervallo* i' *nel sottoalbero sinistro di* x *tale che* $high[i'] = max[left[x]]$. *Quindi* i *non si sovrappone a* i', *né a qualsiasi intervallo* i'' *del sottoalbero destro di* x, *dato che* $low[i'] \leq low[i'']$.

condizione della linea 3 si ha che $max[left[x]] \geq low[i]$. Inoltre per la definizione del campo max, nel sottoalbero sinistro di x ci deve essere un intervallo i' tale che

$$\begin{aligned} high[i'] &= max[left[x]] \\ &\geq low[i]. \end{aligned}$$

La figura 15.5(b) illustra questa situazione. Dato che i ed i' non si sovrappongono e dato che non è vero che $high[i'] < low[i]$, allora per la tricotomia degli intervalli segue che $high[i] < low[i']$. Gli alberi di intervalli hanno il valore delle chiavi uguale agli estremi sinistri degli intervalli e quindi la proprietà dell'albero di ricerca implica che per qualsiasi intervallo i'' del sottoalbero destro di x,

$$\begin{aligned} high[i] &< low[i'] \\ &\leq low[i''], \end{aligned}$$

Per la tricotomia degli intervalli, i e i'' non si sovrappongono. ■

Il Teorema 15.2 garantisce che se la procedura INTERVAL-SEARCH continua su uno dei figli di x e non trova nessun intervallo che si sovrappone a quello dato, allora una ricerca che parta dall'altro figlio di x sarebbe ugualmente vana.

Esercizi

15.3-1 Scrivere lo pseudocodice per LEFT-ROTATE che operi sui nodi di un albero di intervalli ed aggiorni i campi max in tempo $O(1)$.

15.3-2 Riscrivere il codice per INTERVAL-SEARCH in modo tale che operi correttamente quando si assuma che tutti gli intervalli siano aperti.

15.3-3 Descrivere un algoritmo efficiente che, dato un intervallo i, restituisce l'intervallo che si sovrappone a i, che abbia il più piccolo estremo sinistro, oppure NIL se un tale intervallo non esiste.

15.3-4 Dati un albero di intervalli T ed un intervallo i, descrivere come si possano elencare in tempo $O(\min(n, k\lg n))$, tutti gli intervalli di T che si sovrappongono a i, dove k è il numero degli intervalli dell'elenco di output. (*Opzionale*: trovare una soluzione che non modifichi l'albero.)

15.3-5 Descrivere possibili modifiche da apportare alle procedure per gli alberi di intervalli in modo da offrire l'operazione INTERVAL-SEARCH-EXACTLY(T, i), che restituisce il puntatore ad un nodo x dell'albero di intervalli T tale che $low[int[x]] = low[i]$ e $high[int[x]] = high[i]$, oppure NIL se T non contiene un tale nodo. Tutte le operazioni, compresa INTERVAL-SEARCH-EXACTLY, dovrebbero essere eseguite in tempo $O(\lg n)$ su un albero di n nodi.

15.3-6 Mostrare come mantenere un insieme dinamico Q di numeri per offrire l'operazione MIN-GAP, che restituisce la differenza tra i due numeri più vicini tra loro in Q. Per esempio, se $Q = \{1, 5, 9, 15, 18, 22\}$ allora MIN-GAP(Q) restituisce $18 - 15 = 3$, dato che 15 e 18 sono i due numeri più vicini tra loro in Q. Rendere quanto più possibile efficienti le operazioni INSERT, DELETE, SEARCH e MIN-GAP ed analizzare i loro tempi di esecuzione.

* *15.3-7* Le basi di dati VLSI comunemente rappresentano un circuito integrato come una lista di rettangoli. Si assuma che ogni rettangolo abbia i lati paralleli agli assi x e y, in modo che la rappresentazione di un rettangolo sia costituita dalle sue coordinate x e y minime e massime. Fornire un algoritmo che decida in tempo $O(n \lg n)$ se un insieme di rettangoli così rappresentati contenga due rettangoli che si sovrappongono. Non è necessario che l'algoritmo elenchi tutte le coppie che si intersecano, ma deve rispondere che esiste una sovrapposizione se un rettangolo ne copre interamente un altro, anche se i loro lati non si intersecano. (*Suggerimento*: far scorrere una retta attraverso l'insieme dei rettangoli.)

Problemi

15-1 Punto di massima sovrapposizione

Si supponga di voler conoscere il ***punto di massima sovrapposizione*** di un insieme di intervalli: un punto che ha il più grande numero di intervalli che gli si sovrappongono nella base di dati. Mostrare come tale punto possa essere aggiornato in modo efficiente mentre gli intervalli vengono inseriti e cancellati.

15-2 Permutazione di Giuseppe Flavio

Il ***problema di Giuseppe Flavio***[1] è definito nel modo seguente. Si supponga che n persone siano disposte in circolo e che sia dato un intero positivo $m \leq n$. Cominciando dalla persona numero 1, si procede intorno al cerchio allontanando ogni m-esima persona. Dopo che una persona è stata allontanata, il conteggio continua con le persone rimaste nel cerchio. Questo procedimento è ripetuto finché tutte le n persone sono state allontanate. L'ordine con il quale le persone sono allontanate dal cerchio rappresenta la ***permutazione di Giuseppe Flavio*** (n, m) degli interi $1, 2, \ldots, n$. Per esempio la permutazione di Giuseppe Flavio $(7, 3)$ è $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

a. Si supponga m costante. Descrivere un algoritmo con tempo $O(n)$ che, dato un intero n, calcola la permutazione di Giuseppe Flavio (n, m).

[1] N.d.R. Il problema è chiamato così perché spesso correlato a un racconto dello storico del primo secolo Giuseppe Flavio.

b. Si supponga m non costante. Descrivere un algoritmo con tempo $O(n \lg n)$ che, dati due interi n e m, calcola la permutazione di Giuseppe Flavio (n, m).

Note al capitolo

Preparata e Shamos [160] descrivono parecchi alberi di intervalli che appaiono nella letteratura. Tra essi i più importanti da un punto di vista teorico sono quelli proposti indipendentemente da H. Edelsbrunner (1980) e E. M. McCreight (1981) che, in una base di dati di n intervalli, permettono di elencare, in tempo $O(k + \lg n)$, tutti i k intervalli che si sovrappongono a un intervallo dato.

IV Tecniche evolute per il progetto e l'analisi di algoritmi

Introduzione

In questa parte verranno presentati tre metodi molto importanti per il progetto e l'analisi di algoritmi efficienti: la programmazione dinamica (Capitolo 16), gli algoritmi greedy (Capitolo 17) e l'analisi ammortizzata (Capitolo 18). Nelle parti precedenti di questo libro sono state presentate diverse tecniche algoritmiche di vasta applicabilità, quali divide-et-impera, la randomizzazione e la soluzione delle ricorrenze. Le tecniche algoritmiche che verranno presentate in questa parte sono più sofisticate di quelle descritte in precedenza e sono essenziali per affrontare in modo efficace numerosi problemi computazionali. I concetti e le tecniche introdotte in questa parte verranno applicate all'analisi ed alla realizzazione di algoritmi che si incontreranno in seguito.

La programmazione dinamica, come il metodo divide-et-impera, risolve problemi computazionali mettendo assieme le soluzioni di un certo numero di sottoproblemi. (In questo contesto con il termine "programmazione" non si intende un programma scritto in un qualche linguaggio di programmazione ma un metodo di soluzione tabulare.)

Come abbiamo descritto nel Capitolo 1, gli algoritmi basati sul metodo divide-et-impera suddividono il problema in un certo numero di sottoproblemi indipendenti, risolvono ricorsivamente i sottoproblemi e infine fondono assieme le soluzioni ottenute per determinare la soluzione del problema originale.

Il metodo della programmazione dinamica, invece, si può applicare anche quando i sottoproblemi non sono indipendenti: in altri termini la soluzione di alcuni sottoproblemi richiede di risolvere sottoproblemi comuni. In questi casi, un algoritmo basato sul metodo divide-et-impera fa più lavoro di quello strettamente necessario dato che risolve più volte i sottoproblemi comuni. Gli algoritmi di programmazione dinamica risolvono ciascun sottoproblema una sola volta e memorizzano la soluzione in una tabella.

La programmazione dinamica generalmente viene adottata per risolvere problemi di ottimizzazione. Questi problemi ammettono diverse soluzioni. A ogni soluzione è associato un valore e si desidera determinare una soluzione con il valore ottimo (il massimo o il minimo valore). Una soluzione con valore ottimo non viene detta *la* soluzione ottima, ma bensì *una* soluzione ottima del problema, dato che possono esistere diverse soluzioni con valore ottimo. Inoltre molte strategie di soluzione di problemi di ottimizzazione richiedono di risolvere più volte lo stesso sottoproblema e la programmazione dinamica diventa efficiente quando uno stesso sottoproblema si ripresenta per più di una sequenza di decisioni: l'idea fondamentale

della programmazione dinamica è di memorizzare le soluzioni di questi sottoproblemi in modo tale che possano essere utilizzate nel caso in cui si richieda di risolvere un sottoproblema risolto in precedenza. Nel Capitolo 16 si vedrà come questa semplice idea permetta di trasformare algoritmi con complessità esponenziale in algoritmi con complessità polinomiale.

Generalmente gli algoritmi per la risoluzione di problemi di ottimizzazione sono costituiti da sequenze di passi elementari in cui, a ogni passo, si presentano un certo numero di scelte alternative. Per alcuni problemi di ottimizzazione, l'uso di tecniche di programmazione dinamica per decidere la scelta migliore risulta inutilmente oneroso; lo stesso risultato potrebbe essere ottenuto con algoritmi più semplici e più efficienti. Gli algoritmi greedy adottano la strategia di prendere quella decisione che, al momento, appare come la migliore. In altri termini, viene sempre presa quella decisione che, localmente, è la decisione ottima, senza preoccuparsi se tale decisione porterà a una soluzione ottima del problema nella sua globalità. In generale, non sempre esiste un algoritmo greedy che determina la soluzione ottima di un qualsiasi problema, tuttavia gli algoritmi greedy possono essere utilizzati per ottenere le soluzioni di alcune classi di problemi.

Un semplice esempio è fornito dal problema del resto con un numero minimo di monete. Questo problema può essere così formulato: si deve minimizzare il numero totale di monete necessarie per dare il resto a un cliente. Una strategia che risolve questo problema consiste nel selezionare ogni volta un certo numero di monete di valore massimo la cui somma non supera l'ammontare di denaro che deve essere ancora restituito.

Esistono numerosi problemi in cui un metodo greedy fornisce una soluzione ottima molto più velocemente di un metodo di programmazione dinamica. Tuttavia non sempre si è in grado di affermare con precisione che una soluzione greedy sarà efficace.

Nel Capitolo 17 verrà introdotta la teoria dei matroidi che spesso può essere molto utile per determinare la bontà di un metodo greedy.

L'analisi ammortizzata è uno strumento per analizzare algoritmi che eseguono sequenze di operazioni identiche. Invece di limitare il costo di una sequenza di operazioni definendo separatamente il limite del costo di ciascuna operazione della sequenza, l'analisi ammortizzata fornisce un limite del costo dell'intera sequenza di operazioni. Il motivo dell'efficacia di questa idea è dato dal fatto che molto spesso è impossibile che tutte le operazioni della sequenza siano eseguite nel loro caso pessimo. Alcune operazioni potranno essere molto costose, mentre altre potranno risultare particolarmente economiche. Tuttavia, l'analisi ammortizzata non è solamente uno strumento di analisi, ma permette anche di ragionare sul progetto di algoritmi; infatti il progetto di un algoritmo e l'analisi del suo tempo di esecuzione sono attività collegate in modo stretto.

Nel Capitolo 18 verranno introdotti tre metodi diversi per effettuare l'analisi ammortizzata di algoritmi. Nel metodo degli aggregati prima si determina un limite superiore $T(n)$ del costo totale di una sequenza di n operazioni. Il costo ammortizzato di una operazione è dato da $T(n)/n$. Nel metodo degli accantonamenti si stabilisce il costo ammortizzato di ogni operazione. Quando si considerano operazioni di tipo diverso, allora può accadere che il costo ammortizzato di una operazione vari a seconda del tipo. Il metodo degli accantonamenti addebita un costo aggiuntivo a determinate operazioni della sequenza: questo addebito deve essere considerato come un "credito prepagato" relativamente a quello specifico oggetto della struttura di dati.

Successivamente, il credito è usato per definire il costo di quelle operazioni a cui altrimenti verrebbe addebitato un costo minore di quello reale. Il metodo del potenziale ha caratteristi-

che simili al metodo degli accantonamenti, nel senso che stabilisce il costo ammortizzato di ogni operazione, sovrastimando, inizialmente, il costo di determinate operazioni; successivamente, questo addebito viene utilizzato per compensare i costi sottostimati. Differentemente dal metodo degli aggregati, dove il credito è associato agli oggetti individuali della struttura di dati, nel metodo del potenziale il credito è più propriamente visto come "l'energia potenziale", o più semplicemente "il potenziale", della struttura di dati.

Programmazione dinamica

16

La programmazione dinamica, come il metodo divide-et-impera, risolve problemi computazionali mettendo assieme le soluzioni di un certo numero di sottoproblemi. (In questo contesto con il termine "programmazione" non si intende un programma scritto in un qualche linguaggio di programmazione ma un metodo di soluzione tabulare.) Come abbiamo descritto nel Capitolo 1, gli algoritmi basati sul metodo divide-et-impera suddividono il problema in un certo numero di sottoproblemi indipendenti, risolvono ricorsivamente i sottoproblemi e infine fondono assieme le soluzioni ottenute per determinare la soluzione del problema originale. Il metodo della programmazione dinamica, invece, si può applicare anche quando i sottoproblemi non sono indipendenti: in altri termini la soluzione di alcuni sottoproblemi richiede di risolvere i medesimi sottoproblemi. In questi casi, un algoritmo basato sul metodo divide-et-impera fa più lavoro di quello strettamente necessario dato che risolve più volte i sottoproblemi comuni. Gli algoritmi di programmazione dinamica risolvono ciascun sottoproblema una sola volta e memorizzano la soluzione in una tabella; in questo modo si evita di calcolare nuovamente la soluzione ogni volta che deve essere risolto lo stesso sottoproblema.

La programmazione dinamica generalmente viene adottata per risolvere ***problemi di ottimizzazione***. Questi problemi ammettono diverse soluzioni. A ogni soluzione è associato un valore e si desidera determinare una soluzione con il valore ottimo (il massimo o il minimo valore). Una soluzione con valore ottimo non viene detta *la* soluzione ottima, ma bensì *una* soluzione ottima del problema, dato che possono esistere diverse soluzioni con valore ottimo.

Lo sviluppo di un algoritmo di programmazione dinamica può essere diviso in quattro fasi o passi:

1. Caratterizzazione della struttura di una soluzione ottima.
2. Definizione ricorsiva del valore di una soluzione ottima.
3. Calcolo del valore di una soluzione ottima con una strategia bottom-up.
4. Costruzione di una soluzione ottima a partire dalle informazioni calcolate.

Le fasi 1-3 definiscono la base di una soluzione di programmazione dinamica di un certo problema. Nel caso in cui si debba determinare solamente il valore di una soluzione ottima, il passo 4 non viene preso in considerazione. Tuttavia, quando è richiesto di costruire una soluzione ottima, allora durante i calcoli del passo 3 vengono memorizzate delle informazioni ausiliarie che facilitano l'esecuzione del passo 4.

Nei paragrafi seguenti utilizzeremo il metodo della programmazione dinamica per risolvere alcuni problemi di ottimizzazione. Nel paragrafo 16.1 affronteremo il problema del prodotto di una sequenza di matrici in modo da minimizzare il numero complessivo di moltiplicazioni scalari. Dopo questo esempio di programmazione dinamica, il paragrafo 16.2 discute le due caratteristiche peculiari che deve possedere un problema affinché la programmazione dinamica risulti una tecnica di soluzione efficiente. Il paragrafo 16.3 descrive la soluzione del problema della più lunga sottosequenza comune di due sequenze. Infine, nel paragrafo 16.4 viene usata la programmazione dinamica per determinare una triangolazione ottima di un poligono convesso; quest'ultimo problema presenta considerevoli e sorprendenti analogie con il problema del prodotto di una sequenza di matrici.

16.1 Prodotto di una sequenza di matrici

Il nostro primo esempio di programmazione dinamica è un algoritmo che risolve il problema del prodotto di una sequenza di matrici. Sia $\langle A_1, A_2, \ldots A_n \rangle$ una sequenza di n matrici, si vuole calcolare il prodotto matriciale

$$A_1 A_2 \cdots A_n \,. \tag{16.1}$$

Possiamo valutare l'espressione (16.1) utilizzando come sottoprogramma l'algoritmo standard del prodotto di due matrici, dopo una opportuna parentesizzazione che elimini le ambiguità sull'ordine dei prodotti tra le matrici della sequenza. Un prodotto di matrici viene detto ***completamente parentesizzato*** se consiste di un'unica matrice, o è il prodotto, delimitato da parentesi, di due prodotti di matrici completamente parentesizzati. Dato che l'operazione di prodotto matriciale è associativa, tutte le parentesizzazioni definiscono la medesima matrice risultato. Per esempio, considerando la sequenza di matrici $\langle A_1, A_2, A_3, A_4 \rangle$, il prodotto matriciale $A_1 A_2 A_3 A_4$ può essere completamente parentesizzato in cinque modi distinti:

$(A_1 (A_2 (A_3 A_4)))$,
$(A_1 ((A_2 A_3) A_4))$,
$((A_1 A_2) (A_3 A_4))$,
$((A_1 (A_2 A_3)) A_4)$,
$(((A_1 A_2) A_3) A_4)$.

La struttura della parentesizzazione di una sequenza di matrici influisce sul costo del calcolo del prodotto matriciale. Prima di tutto determiniamo il costo dell'algoritmo che calcola il prodotto di due matrici. La seguente procedura (in pseudocodice) descrive l'algoritmo standard del prodotto di due matrici. Nel seguito con *rows*[A] e *columns*[A] indichiamo rispettivamente il numero di righe e di colonne di una matrice A.

Due matrici A e B possono essere moltiplicate solo se il numero di colonne di A è uguale al numero di righe di B. Se A è una matrice $p \times q$ e B è una matrice $q \times r$, il risultato della moltiplicazione di A e B è la matrice C di dimensioni $p \times r$. Il tempo necessario per calcolare la matrice C è determinato dal numero delle moltiplicazioni scalari della linea 7, cioè $p\,q\,r$. Nel seguito, il tempo di esecuzione verrà sempre espresso in termini del numero di moltiplicazioni scalari.


```
MATRIX-MULTIPLY(A, B)
1  if columns[A] ≠ rows[B]
2     then error "dimensioni non compatibili"
3     else for i ← 1 to rows[A]
4              do for j ← 1 to columns[B]
5                     do C[i, j] ← 0
6                        for k ← 1 to columns[A]
7                            do C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8          return C
```

Per mettere in evidenza che parentesizzazioni differenti del medesimo prodotto matriciale danno origine a costi differenti, consideriamo una sequenza di tre matrici $\langle A_1, A_2, A_3 \rangle$. Supponiamo che le dimensioni delle matrici siano, rispettivamente, 10×100, 100×5 e 5×50. Se si calcola il prodotto della sequenza con la parentesizzazione $((A_1 A_2) A_3)$, allora per calcolare la matrice di dimensioni 10×5 risultante dal prodotto $A_1 A_2$ si eseguono $10 \cdot 100 \cdot 5 = 5000$ moltiplicazioni scalari. A questo numero dobbiamo sommare le $10 \cdot 5 \cdot 50 = 2500$ moltiplicazioni scalari necessarie per moltiplicare la matrice $A_1 A_2$ con la matrice A_3. Pertanto abbiamo un totale di 7500 moltiplicazioni scalari. Invece, se il prodotto della sequenza viene calcolato con la parentesizzazione $(A_1(A_2 A_3))$, sono necessarie $100 \cdot 5 \cdot 50 = 25000$ moltiplicazioni scalari per calcolare la matrice $A_2 A_3$ di dimensioni 100×50. A questo numero dobbiamo aggiungere $10 \cdot 100 \cdot 50 = 50000$, ovvero il numero delle moltiplicazioni scalari necessarie per moltiplicare la matrice A_1 con la matrice $A_2 A_3$. Pertanto abbiamo un totale di 75000 moltiplicazioni scalari. Risulta evidente che il calcolo del prodotto della sequenza di matrici è 10 volte più veloce se viene effettuato considerando la prima parentesizzazione.

Il ***problema del prodotto di una sequenza di matrici*** può essere formulato nel modo seguente: data una sequenza di n matrici $\langle A_1, A_2, \ldots, A_n \rangle$, dove la generica matrice A_i ha dimensioni $p_{i-1} \times p_i$ con $i = 1, 2, \ldots, n$, si deve determinare una struttura di parentesizzazione completa del prodotto $A_1 A_2 \cdots A_n$ che minimizzi il numero complessivo di moltiplicazioni scalari.

Determinazione del numero di parentesizzazioni

Prima di presentare la soluzione, con un algoritmo di programmazione dinamica, del problema del prodotto di una sequenza di matrici, vogliamo mostrare che il controllo esaustivo di tutte le possibili parentesizzazioni non fornisce un algoritmo di risoluzione efficiente. Con $P(n)$ indichiamo il numero delle diverse parentesizzazioni di una sequenza di n matrici. Dato che una parentesizzazione di una sequenza di matrici può essere sempre ottenuta combinando assieme la parentesizzazione delle prime k matrici con la parentesizzazione delle rimanenti matrici della sequenza originale, per un generico $k = 1, 2, \ldots, n-1$, abbiamo la seguente equazione di ricorrenza:

$$P(n) = \begin{cases} 1 & \text{se } n = 1 \,, \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{se } n \geq 2 \,. \end{cases}$$

Il Problema 13-4 chiedeva di dimostrare che la soluzione di questa equazione di ricorrenza è data dalla sequenza dei ***numeri Catalani***

$$P(n) = C(n-1)\,,$$

dove

$$\begin{aligned} C(n) &= \frac{1}{n+1}\binom{2n}{n} \\ &= \Omega(4^n/n^{3/2})\,. \end{aligned}$$

Il numero di soluzioni, quindi, è esponenziale in n; pertanto per determinare la parentesizzazione ottima di una sequenza di matrici il metodo di soluzione brutale della ricerca esaustiva risulta particolarmente inefficiente.

Struttura della parentesizzazione ottima

Il primo passo del paradigma della programmazione dinamica è la caratterizzazione della struttura della soluzione ottima. Nel caso del problema del prodotto di una sequenza di matrici questo passo può essere realizzato nel modo seguente. Per ragioni di convenienza di notazione, indichiamo con $A_{i..j}$ la matrice risultato della valutazione del prodotto matriciale $A_i A_{i+1} \cdots A_j$. Una parentesizzazione ottima del prodotto $A_1 A_2 \cdots A_n$ divide il problema nei due sottoproblemi del prodotto delle prime k matrici e del prodotto delle rimanenti matrici, per un qualche intero k tale che $1 \le k < n$. In altri termini, prima si determinano le matrici $A_{1..k}$ e $A_{k+1..n}$, per un determinato valore di k. Infine, la soluzione finale $A_{1..n}$ è il risultato del prodotto delle due matrici ottenute in precedenza. Il costo della parentesizzazione ottima è dato dalla somma del costo del calcolo della matrice $A_{1..k}$, con il costo del calcolo di $A_{k+1..n}$ a cui si deve aggiungere il costo del prodotto delle due matrici.

A questo punto vogliamo mettere in rilievo una proprietà cruciale del problema che stiamo esaminando. La parentesizzazione di $A_1 A_2 \cdots A_k$ nella parentesizzazione ottima di $A_1 A_2 \cdots A_n$ deve necessariamente essere una parentesizzazione *ottima* di $A_1 A_2 \cdots A_k$. Il perché della validità di questa proprietà è presto detto. Se esistesse una strategia meno costosa per ottenere una parentesizzazione di $A_1 A_2 \cdots A_k$ allora sarebbe possibile ottenere una parentesizzazione di $A_1 A_2 \cdots A_n$ con un costo inferiore di quella ottima: è sufficiente sostituire la parentesizzazione di $A_1 A_2 \cdots A_k$ nella parentesizzazione ottima di $A_1 A_2 \cdots A_n$. Chiaramente questo ragionamento porta a una contraddizione. È facile vedere che una proprietà analoga vale anche per la parentesizzazione di $A_{k+1} A_{k+2} \cdots A_n$ nella parentesizzazione ottima di $A_1 A_2 \cdots A_n$: deve necessariamente essere una parentesizzazione ottima di $A_{k+1} A_{k+2} \cdots A_n$.

Pertanto, una soluzione ottima del problema del prodotto di una sequenza di matrici contiene al suo interno le soluzioni ottime dei due sottoproblemi. L'esistenza di sottostrutture ottime all'interno di una soluzione ottima, come vedremo nel paragrafo 16.2, è uno dei segni distintivi e dei criteri di scelta per l'applicazione delle tecniche della programmazione dinamica.

Soluzione ricorsiva

Il secondo passo del paradigma della programmazione dinamica consiste nella definizione del valore di una soluzione ottima ricorsivamente in termini dei valori delle soluzioni ottime dei sottoproblemi. Nel caso del problema del prodotto di una sequenza di matrici, i

sottoproblemi consistono nel problema di determinare il costo minimo di una parentesizzazione di $A_i A_{i+1} \cdots A_j$ per $1 \le i \le j \le n$. Sia $m[i, j]$ il minimo numero di moltiplicazioni scalari necessarie per calcolare la matrice $A_{i..j}$; pertanto il costo minimo del calcolo della matrice prodotto $A_{1..n}$ è dato da $m[1, n]$.

Possiamo definire $m[i, j]$ ricorsivamente nel modo seguente. Se $i = j$ allora la sequenza consiste di una singola matrice, pertanto dato che $A_{i..i} = A_i$ *non* è necessario eseguire alcuna operazione scalare per calcolare il prodotto. Da questo consegue che $m[i, i] = 0$ per $i = 1, 2, \ldots, n$. Nel caso in cui $i < j$, il calcolo di $m[i, j]$ tiene in considerazione la struttura di una soluzione ottima determinata al passo 1. Infatti, assumiamo che la parentesizzazione ottima sia tale che il prodotto $A_i A_{i+1} \cdots A_j$ sia ottenuto combinando assieme i prodotti $A_{i..k}$ e $A_{k+1..j}$, con $i \le k < j$. È facile vedere che il valore di $m[i, j]$ è ottenuto sommando i costi minimi del calcolo dei sottoprodotti $A_{i..k}$ e $A_{k+1..j}$ con il costo del prodotto di queste due matrici. Dato che il calcolo del prodotto tra $A_{i..k}$ e $A_{k+1..j}$ richiede l'esecuzione di $p_{i-1} p_k p_j$ moltiplicazioni scalari ne consegue che

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\,.$$

L'equazione ricorsiva precedente assume che il valore di k sia noto; ma chiaramente questo non è il caso. Tuttavia, i possibili valori di k sono $j - i$ dato che $k = i, i+1, \ldots, j-1$. Poiché la parentesizzazione ottima deve necessariamente assegnare a k uno di questi valori, il valore migliore da assegnare a k è dato dal valore che minimizza l'equazione precedente. Pertanto la definizione ricorsiva del costo minimo della parentesizzazione ottima del prodotto $A_i A_{i+1} \cdots A_j$ diventa

$$m[i, j] = \begin{cases} 0 & \text{se } i = j\,, \\ \min\limits_{i \le k < j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{se } i < j\,. \end{cases} \tag{16.2}$$

Il valore di $m[i, j]$ definisce i costi delle soluzioni ottime dei sottoproblemi. Per tenere traccia del modo in cui costruire una soluzione ottima, definiamo $s[i, j]$ quel valore k per cui, dividendo il prodotto $A_i A_{i+1} \cdots A_j$, si ottiene una parentesizzazione ottima. In altri termini, $s[i, j]$ è quel valore k tale che $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j$.

Calcolo dei costi ottimi

Alla luce delle considerazioni precedenti è abbastanza facile definire un algoritmo ricorsivo, basato sulla equazione di ricorrenza (16.2), che calcola il costo minimo $m[1, n]$ del prodotto $A_1 A_2 \cdots A_n$. Tuttavia, come vedremo nel paragrafo 16.2, questo algoritmo è esponenziale nel tempo, come l'algoritmo ingenuo che controlla in modo esaustivo tutte le possibili parentesizzazioni del prodotto.

Questo è il punto per fare una osservazione decisiva. Nel problema del prodotto di una sequenza di matrici abbiamo un numero relativamente basso di sottoproblemi: un problema per ogni possibile scelta di i e j, con $1 \le i \le j \le n$, per un totale di $\binom{n}{2} + n = \Theta(n^2)$.

Un algoritmo ricorsivo, nei diversi cammini del suo albero di ricorsione, può richiedere di risolvere più di una volta lo stesso sottoproblema. La caratteristica di avere sottoproblemi comuni è il secondo segno distintivo e criterio di applicabilità della programmazione dinamica.

Invece di calcolare in modo ricorsivo la soluzione dell'equazione di ricorrenza (16.2), procediamo con la terza fase del paradigma della programmazione dinamica: calcoliamo il costo della soluzione ottima con una modalità bottom-up. La procedura descritta di seguito assume che la generica matrice A_i sia di dimensioni $p_{i-1} \times p_i$, con $i = 1, 2, \ldots, n$. Il dato di ingresso della procedura è una sequenza $p = \langle p_0, p_1, \ldots, p_n \rangle$, dove $length[p] = n + 1$. Infine la procedura utilizza una tabella ausiliaria $m[1 \ldots n, 1 \ldots n]$ per memorizzare i costi $m[i, j]$ ed una tabella ausiliaria $s[1 \ldots n, 1 \ldots n]$ per memorizzare il valore dell'indice k per cui $m[i, j]$ è un costo ottimo.

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3      do m[i, i] ← 0
 4  for l ← 2 to n
 5      do for i ← 1 to n − l + 1
 6             do  j ← i + l − 1
 7                 m[i, j] ← ∞
 8                 for k ← i to j − 1
 9                     do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                        if q < m[i, j]
11                           then m[i, j] ← q
12                                s[i, j] ← k
13  return m ed s
```

L'algoritmo costruisce la tabella m nello stesso ordine in cui viene risolto il problema della parentesizzazione su una sequenza di matrici di lunghezza crescente. L'equazione (16.2) mostra che il costo $m[i, j]$ del calcolo del prodotto di una sequenza di $j - i + 1$ matrici dipende solamente dal costo del calcolo del prodotto di sequenze di matrici di una lunghezza minore di $j - i + 1$. In altri termini, per $k = i, i + 1, \ldots, j - 1$, la matrice $A_{i \ldots k}$ è il risultato del prodotto di $k - i + 1 < j - i + 1$ matrici, e la matrice $A_{k+1 \ldots j}$ è il risultato del prodotto di $j - k < j - i + 1$ matrici.

L'algoritmo per prima cosa (linee 2-3) esegue l'istruzione $m[i, i] \leftarrow 0$ per $i = 1, 2, \ldots, n$ (il costo minimo per sequenze di lunghezza 1). In seguito, durante l'esecuzione del ciclo formato dalle istruzioni delle linee 4-12, viene utilizzata l'equazione di ricorrenza (16.2) per calcolare il valore $m[i, i + 1]$, per $i = 1, 2, \ldots, n - 1$ (il costo minimo per sequenze di lunghezza $l = 2$). Quando il ciclo viene eseguito una seconda volta si calcola il valore $m[i, i + 2]$, per $i = 1, 2, \ldots, n - 2$ (il costo minimo per sequenze di lunghezza $l = 3$) e così via. A ogni passo del ciclo il costo $m[i, j]$ calcolato nelle linee 9-12 dipende solamente dai valori della tabella $m[i, k]$ e $m[k + 1, j]$ calcolati ai passi precedenti.

La figura 16.1 descrive il risultato di questa procedura su una sequenza di $n = 6$ matrici. È facile vedere che viene utilizzata solo quella parte della tabella che rimane al di sopra della diagonale principale: il valore $m[i, j]$ è definito solo quando $i \leq j$. Nella figura la diagonale principale della tabella è disposta orizzontalmente e la sequenza delle matrici è elencata subito sotto la figura. Per come è disposta la tabella nella figura 16.1, si può notare che il costo minimo $m[i, j]$ del prodotto di una sequenza di matrici $A_i A_{i+1} \cdots A_j$ può essere trovato sopra

il punto di intersezione delle linee diagonali che partono da A_i e da A_j. Si noti, inoltre, che ogni linea orizzontale della tabella contiene i valori relativi a sequenze di matrici della stessa lunghezza. La procedura MATRIX-CHAIN-ORDER determina le righe della tabella con una modalità bottom-up, mentre gli elementi di ogni riga vengono calcolati da sinistra verso destra. Ogni elemento $m[i, j]$ dipende dalla valutazione del prodotto scalare $p_{i-1} p_k p_j$ e dagli elementi della tabella $m[i, k]$ e $m[k + 1, j]$, per $k = i, i + 1, \ldots, j - 1$.

Da una semplice analisi della struttura dei cicli della procedura MATRIX-CHAIN-ORDER si vede facilmente che l'algoritmo trova una soluzione in tempo $O(n^3)$. I cicli sono annidati uno dentro l'altro, con tre livelli di annidamento, inoltre n è il numero dei valori che possono assumere le variabili di controllo dei cicli presenti nella procedura (le variabili i, l e k). Nell'Esercizio 16.1-3 si richiede di mostrare che il tempo di esecuzione di questo algoritmo è $\Omega(n^3)$.

Inoltre, la quantità di memoria necessaria per memorizzare le tabelle m e s è $\Theta(n^2)$. In conclusione, la procedura MATRIX-CHAIN-ORDER si dimostra molto più efficiente dell'algoritmo esponenziale che controlla in modo esaustivo tutte le possibili parentesizzazioni.

Figura 16.1 *Le tabelle m e n calcolate dalla procedura* MATRIX-CHAIN-ORDER *nel caso in cui n = 6 e le dimensioni delle matrici siano:*

matrice	dimensione
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Le due tabelle sono posizionate in modo tale che la diagonale principale è disposta orizzontalmente (nella figura). Nella tabella m viene utilizzata solo la diagonale principale e il triangolo superiore della tabella; mentre nella tabella s viene utilizzato solo il triangolo sopra la diagonale. Il numero minimo di moltiplicazioni scalari necessarie per effettuare il prodotto delle sei matrici risulta $m[1, 6] = 15125$. *Le coppie di elementi della tabella che hanno lo stesso colore, leggermente più chiaro rispetto agli altri elementi della tabella, sono quelle che vengono considerate nella linea 9 durante il calcolo di*

$$m[2, 5] = \min \begin{cases} m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 , \\ m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 , \\ m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

$$= 7125 .$$

Costruzione di una soluzione ottima

Nel paragrafo precedente, abbiamo mostrato che la procedura MATRIX-CHAIN-ORDER determina il numero ottimo di moltiplicazioni scalari necessarie per calcolare il prodotto di una sequenza di matrici; tuttavia la procedura non mostra direttamente quale sia la reale strategia di esecuzione dei prodotti matriciali. Il quarto passo del paradigma della programmazione dinamica ha il compito di definire la strategia che permette di costruire una soluzione ottima del problema; la strategia dipende dalle informazioni sulle soluzioni ottime dei sottoproblemi calcolate nelle fasi precedenti.

Nel nostro esempio, la tabella $s[1..n, 1..n]$ contiene tutte le informazioni necessarie per determinare la migliore strategia di calcolo dei prodotti matriciali. Un generico elemento $s[i, j]$ della tabella contiene quel valore k per cui la parentesizzazione ottima di $A_i A_{i+1} \cdots A_j$ è ottenuta combinando assieme le parentesizzazioni ottime di $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$. Questa informazione ci assicura che l'ultimo prodotto matriciale nel calcolo ottimo di $A_{1..n}$ è dato dal prodotto $A_{1..s[1,n]} A_{s[1,n]+1..n}$. Chiaramente questi prodotti matriciali possono essere calcolati in modo ricorsivo: il valore $s[1, s[1, n]]$ definisce solamente l'ultimo prodotto matriciale nel calcolo di $A_{1..s[1,n]}$. Analogamente, $s[s[1, n] + 1, n]$ definisce l'ultimo prodotto matriciale nel calcolo di $A_{s[1,n]+1..n}$. La seguente procedura ricorsiva calcola la matrice $A_{1..n}$ risultato del prodotto di una sequenza di matrici. I parametri della procedura sono la sequenza di matrici $A = \langle A_1 \ldots A_n \rangle$, la tabella s costruita dalla procedura MATRIX-CHAIN-ORDER e gli indici i e j. Si assume che la chiamata iniziale della procedura sia MATRIX-CHAIN-MULTIPLY(A, s, 1, n).

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
  if j > i
      then X ← MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
           Y ← MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
           return MATRIX-MULTIPLY(X, Y)
      else return A_i
```

Nell'esempio di figura 16.1, la chiamata della procedura MATRIX-CHAIN-MULTIPLY(A, s, 1, 6) calcola il prodotto della sequenza di matrici secondo la parentesizzazione

$$((A_1 (A_2 A_3))((A_4 A_5) A_6)) \tag{16.3}$$

Esercizi

16.1-1 Si determini una parentesizzazione ottima del prodotto di una sequenza di matrici di dimensioni $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

16.1-2 Si definisca un algoritmo efficiente, PRINT-OPTIMAL-PARENS, per stampare la parentesizzazione ottima di una sequenza di matrici. La procedura PRINT-OPTIMAL-PARENS ha come solo parametro la tabella s costruita dalla procedura MATRIX-CHAIN-ORDER. Si discutano le caratteristiche e le proprietà dell'algoritmo proposto.

16.1-3 Sia $R(i, j)$ il numero di volte che la procedura MATRIX-CHAIN-ORDER accede all'elemento $m[i, j]$ nel calcolo degli altri elementi della tabella m. Si dimostri che il numero totale di accessi alla tabella m è dato da

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3} .$$

(*Suggerimento*: la soluzione del problema può essere facilitata dall'utilizzo dell'equazione $\sum_{i=1}^{n} i^2 = n(n + 1)(2n + 1)/6$.)

16.1-4 Si mostri che la parentesizzazione completa di una espressione con n elementi ha esattamente $n - 1$ coppie di parentesi.

16.2 Elementi di programmazione dinamica

Nel paragrafo precedente abbiamo mostrato solamente un esempio di applicazione della tecnica della programmazione dinamica: a questo punto è naturale chiedersi quale sia la classe di problemi che possono essere risolti con questa tecnica. In questo paragrafo esamineremo le due proprietà basilari, sottostruttura ottima e sottoproblemi comuni, che un problema di ottimizzazione deve possedere affinché si possa applicare la tecnica della programmazione dinamica. Inoltre, presenteremo una variante della tecnica della programmazione dinamica, detta ricorsione con memorizzazione (*memoization*), che si avvantaggia della proprietà dei sottoproblemi comuni.

Sottostruttura ottima

Il primo passo di risoluzione di un problema di ottimizzazione mediante la tecnica della programmazione dinamica consiste nella caratterizzazione della struttura di una soluzione ottima. Diciamo che un problema presenta una ***sottostruttura ottima*** nel caso in cui una soluzione ottima del problema contiene al suo interno soluzioni ottime dei sottoproblemi. Ogni qualvolta un problema presenta una sottostruttura ottima, questo potrebbe essere un buon indizio dell'applicabilità della tecnica della programmazione dinamica. (Tuttavia, questo potrebbe volere indicare anche l'applicabilità di una strategia greedy: vedi Capitolo 17.)

Nel paragrafo 16.1 abbiamo mostrato che il problema del prodotto di una sequenza di matrici presenta una sottostruttura ottima. Infatti una parentesizzazione ottima di $A_1 A_2 \cdots A_n$ contiene soluzioni ottime dei problemi che trovano le parentesizzazioni di $A_1 A_2 \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_n$. La tecnica che abbiamo adottato per dimostrare che i sottoproblemi presentano una soluzione ottima è una tecnica standard. Si assume che esista una soluzione migliore del sottoproblema e si dimostra che questa ipotesi porta a contraddire l'ipotesi di ottimalità del problema originale.

Spesso, la sottostruttura ottima di un problema indica l'esistenza di un insieme di sottoproblemi (detto *spazio dei sottoproblemi*) a cui si può applicare la tecnica della programmazione dinamica. Generalmente, per un certo problema esistono diverse classi di sottoproblemi che possono essere considerate come le classi di sottoproblemi "naturali". Per esempio, lo spazio dei sottoproblemi che abbiamo considerato nel caso del prodotto di una sequenza di matrici include tutti i sottoproblemi per le sottosequenze della sequenza di ingresso. Avremmo potuto anche scegliere come spazio dei sottoproblemi sequenze arbitrarie di matrici costruite a partire dalla sequenza di ingresso. Tuttavia quest'ultimo spazio dei sottoproblemi è troppo grande: un algoritmo di programmazione dinamica basato su questo spazio dei sottoproblemi dovrebbe risolvere un numero di problemi maggiore di quello strettamente necessario.

Nel caso della programmazione dinamica, un buon metodo di definizione dello spazio dei sottoproblemi consiste nell'analizzare la sottostruttura ottima del problema e di ogni sottoproblema. Per esempio, dopo avere esaminato la sottostruttura ottima del problema del prodotto di una sequenza di matrici, possiamo applicare lo stesso meccanismo per esaminare le strutture ottime dei sottoproblemi, dei sottosottoproblemi e così via. Al termine di questa analisi scopriamo che tutti i sottoproblemi non sono altro che prodotti di sottosequenze di $\langle A_1, A_2, \ldots, A_n \rangle$. Pertanto, lo spazio dei sottoproblemi più naturale e maneggevole da considerare è dato dalle sequenze di matrici della forma $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, con $1 \le i \le j \le n$.

Sottoproblemi comuni

La seconda proprietà, che un problema di ottimizzazione deve soddisfare affinché si possa applicare la tecnica della programmazione dinamica, è che lo spazio delle soluzioni sia "piccolo", nel senso che un algoritmo di risoluzione ricorsivo risolve più di una volta gli stessi sottoproblemi, piuttosto che continuare a generare nuovi sottoproblemi. Generalmente, il numero totale di sottoproblemi distinti è polinomiale nelle dimensioni dei dati in ingresso. Diciamo che un problema di ottimizzazione ha ***sottoproblemi comuni*** nel caso in cui un algoritmo ricorsivo richiede di risolvere più di una volta lo stesso problema. La differenza principale con la tecnica divide-et-impera è che quest'ultima tecnica, a ogni passo ricorsivo, genera un problema completamente nuovo. Invece, gli algoritmi di programmazione dinamica si avvantaggiano della proprietà dei sottoproblemi comuni. Ogni sottoproblema è risolto una sola volta e le soluzioni di tutti i sottoproblemi sono memorizzate in una tabella. Quando si presenta nuovamente il medesimo sottoproblema la sua soluzione viene trovata con un semplice accesso alla tabella. L'accesso alla tabella richiede sempre un tempo costante.

Per illustrare la proprietà dei sottoproblemi comuni, consideriamo nuovamente il problema del prodotto di una sequenza di matrici. Facendo riferimento alla figura 16.1 si può notare come la procedura MATRIX-CHAIN-ORDER riutilizzi le soluzioni dei sottoproblemi delle righe inferiori durante la risoluzione dei sottoproblemi delle righe superiori. Per esempio, l'elemento $m[3, 4]$ viene utilizzato per ben quattro volte: durante il calcolo di $m[2, 4]$, $m[1, 4]$, $m[3, 5]$ e $m[3, 6]$. Se $m[3, 4]$ fosse calcolato nuovamente ogni volta, invece che essere ottenuto con un semplice accesso alla tabella, il tempo di esecuzione aumenterebbe notevolmente. Per vedere più dettagliatamente questo fatto si consideri la seguente (non efficiente) procedura ricorsiva che determina $m[i, j]$, il minimo numero di moltiplicazioni scalari necessarie per calcolare il prodotto matriciale $A_{i..j} = A_i A_{i+1} \ldots A_j$. La procedura è definita in termini dell'equazione di ricorrenza (16.2).

Figura 16.2 *L'albero di ricorsione della chiamata di* RECURSIVE-MATRIX-CHAIN(p, 1, 4). *Ogni nodo contiene i parametri i e j. La parte della computazione illustrata dai sottoalberi in grigio è sostituita da un singolo accesso in tabella nella chiamata di* MEMOIZED-MATRIX-CHAIN(p, 1, 4).

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i = j
2     then return 0
3  m[i, j] ← ∞
4  for k ← i to j − 1
5      do  q ← RECURSIVE-MATRIX-CHAIN(p, i, k)
                 + RECURSIVE-MATRIX-CHAIN (p, k + 1, j) + p_{i−1} p_k p_j
6          if q < m[i, j]
7             then m[i, j] ← q
8  return m[i, j]
```

La figura 16.2 mostra l'albero di ricorsione prodotto dalla chiamata della procedura RECURSIVE-MATRIX-CHAIN(p, 1, 4). Ogni nodo dell'albero ha come etichetta i valori dei parametri i e j. Si noti come alcune coppie di valori siano ripetute.

Infatti, si può dimostrare che il tempo di esecuzione $T(n)$ del calcolo di $m[1, n]$ con questa procedura ricorsiva è esponenziale in n. Per semplicità assumiamo che l'esecuzione delle istruzioni delle linee 1-2 e 6-7 richieda almeno una unità di tempo. Da un'analisi della procedura si ottiene la seguente equazione di ricorrenza

$$T(1) \ge 1 ,$$

$$T(n) \ge 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{per } n > 1 .$$

Si noti come un generico termine $T(i)$, per $i = 1, 2, \ldots, n-1$, si presenti una volta come $T(k)$ e un'altra volta come $T(n-k)$; inoltre, combinando assieme le $n-1$ occorrenze di "1" nella sommatoria con l'"1" fuori dalla sommatoria possiamo riscrivere l'equazione di ricorrenza come

$$T(n) \ge 2 \sum_{i=1}^{n-1} T(i) + n . \tag{16.4}$$

Mediante il metodo di sostituzione dimostriamo che $T(n) = \Omega(2^n)$. In particolare, dimostriamo per induzione che $T(n) \geq 2^{n-1}$, per ogni $n \geq 2$. La base dell'induzione è immediata dato che $T(1) \geq 1 = 2^0$. Per $n \geq 2$ il passo induttivo risulta

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^n - 2) + n \\ &\geq 2^{n-1} . \end{aligned}$$

Questo conclude la dimostrazione. In conclusione, il numero totale delle operazioni eseguite da una chiamata della RECURSIVE-MATRIX-CHAIN$(p, 1, n)$ risulta essere esponenziale in n.

Si confronti il precedente algoritmo ricorsivo top-down con l'algoritmo di programmazione dinamica bottom-up. Quest'ultimo è più efficiente dato che si avvantaggia della proprietà dei sottoproblemi comuni. Esistono solamente $\Theta(n^2)$ sottoproblemi distinti e l'algoritmo di programmazione dinamica li risolve ciascuno una sola volta. Al contrario, l'algoritmo ricorsivo risolve ciascun sottoproblema ogni qualvolta si ripresenta nell'albero di ricorsione. Se nell'albero di ricorsione della soluzione ricorsiva naturale di un problema lo stesso sottoproblema si ripresenta più volte, allora è una buona idea domandarsi se non sia possibile applicare la tecnica della programmazione dinamica.

Ricorsione con memorizzazione

Esiste una variante della programmazione dinamica che, pur adottando una strategia di tipo top-down, spesso presenta la stessa efficienza delle soluzioni algoritmiche di programmazione dinamica. L'idea è di memorizzare il naturale, ma inefficiente, algoritmo ricorsivo. Come in tutte le soluzioni di programmazione dinamica anche in questo caso le soluzioni dei sottoproblemi vengono memorizzate in una tabella, ma la novità principale di questa alternativa è data dalla strategia di controllo della costruzione della tabella che è sostanzialmente quella definita dall'algoritmo ricorsivo.

Un algoritmo ***ricorsivo con memorizzazione*** (*memoized*) utilizza una tabella in cui vengono memorizzate le soluzioni di ogni sottoproblema. Ogni elemento della tabella inizialmente assume un valore speciale per indicare che il valore di quell'elemento non è ancora stato calcolato. Quando, durante l'esecuzione dell'algoritmo ricorsivo, si richiede di risolvere per la prima volta un generico sottoproblema, allora si calcola la soluzione e la si memorizza nella tabella. Quando viene richiesto di risolvere nuovamente lo stesso sottoproblema, semplicemente si accede alla tabella e si restituisce, come risultato, il valore memorizzato.[1]

[1] Questa soluzione presuppone che i parametri di tutti i possibili sottoproblemi siano noti a priori ed inoltre che esista una corrispondenza tra le posizioni della tabella ed i sottoproblemi. Una soluzione alternativa consiste nell'utilizzo di tecniche hash dove i parametri dei sottoproblemi vengono utilizzati come chiavi.

La seguente procedura è una versione con memorizzazione della procedura RECURSIVE-MATRIX-CHAIN.

```
MEMOIZED-MATRIX-CHAIN(p)
1  n ← length[p] − 1
2  for i ← 1 to n
3        do for  j ← i to n
4                do m[i, j] ← ∞
5  return LOOKUP-CHAIN(p, 1, n)

LOOKUP-CHAIN(p, i, j)
1  if m[i, j] < ∞
2        then return m[i, j]
3  if i = j
4        then m[i, j] ← 0
5        else for k ← i to j − 1
6                do q ← LOOKUP-CHAIN(p, i, k)
                        + LOOKUP-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7                if q < m[i, j]
8                        then m[i, j] ← q
9  return m[i, j]
```

La procedura MEMOIZED-MATRIX-CHAIN, allo stesso modo della procedura MATRIX-CHAIN-ORDER, utilizza una tabella $m[1 \ldots n, 1 \ldots n]$ dove un generico elemento $m[i, j]$ rappresenta il numero minimo di moltiplicazioni scalari necessarie per calcolare la matrice $A_{i \ldots j}$. Ogni elemento della tabella inizialmente prende il valore ∞ per indicare che il corrispondente numero di moltiplicazioni scalari deve essere ancora calcolato. Se, al momento della chiamata della procedura LOOKUP-CHAIN(p, i, j), $m[i, j] < \infty$, allora la procedura semplicemente restituisce come risultato il costo $m[i, j]$ calcolato in precedenza (linea 2). Altrimenti, come nel caso della procedura RECURSIVE-MATRIX-CHAIN, si calcola il costo, lo si memorizza in $m[i, j]$ e poi si restituisce il valore calcolato. (Conviene usare il valore speciale ∞ per indicare che l'elemento della tabella non è significativo dato che quello stesso valore è utilizzato alla linea 3 della procedura RECURSIVE-MATRIX-CHAIN come valore iniziale di $m[i, j]$.) In conclusione, la chiamata della procedura LOOKUP-CHAIN(p, i, j) restituisce sempre il valore $m[i, j]$, ma questo valore viene calcolato effettivamente solo la prima volta in cui la procedura è chiamata con i parametri i e j.

La figura 16.2 mostra come la procedura MEMOIZED-MATRIX-CHAIN sia più efficiente della procedura RECURSIVE-MATRIX-CHAIN. I sottoalberi più scuri rappresentano quei valori che sono ottenuti mediante un accesso alla tabella e non calcolati direttamente.

Allo stesso modo dell'algoritmo di programmazione dinamica MATRIX-CHAIN-ORDER, la procedura MEMOIZED-MATRIX-CHAIN viene eseguita in tempo $O(n^3)$. Ognuno dei $\Theta(n^2)$ elementi della tabella è inizializzato una sola volta nella linea 4 della procedura MEMOIZED-MATRIX-CHAIN e gli viene assegnato il valore corretto grazie a una sola chiamata della procedura LOOKUP-CHAIN. Ognuna di queste $\Theta(n^2)$ chiamate della procedura LOOKUP-CHAIN impiega tempo $O(n)$, se si esclude il tempo necessario a calcolare gli altri elementi della

tabella, pertanto, in totale si impiega tempo $O(n^3)$. Quindi, la tecnica della ricorsione con memorizzazione trasforma un algoritmo $\Omega(2^n)$ in un algoritmo $O(n^3)$.

In conclusione, il problema del prodotto di una sequenza di matrici può essere risolto in un tempo $O(n^3)$ sia da un algoritmo top-down, che impiega una tecnica di ricorsione con memorizzazione, che da un algoritmo bottom-up di programmazione dinamica. Entrambe le soluzioni si avvantaggiano della proprietà dei sottoproblemi comuni: ci sono solamente $\Theta(n^2)$ sottoproblemi distinti e le due procedure calcolano la soluzione di ogni sottoproblema esattamente una sola volta. Senza l'uso della memorizzazione, l'algoritmo ricorsivo naturale richiederebbe un tempo esponenziale dato che lo stesso sottoproblema verrebbe risolto più volte.

In generale, se tutti i sottoproblemi devono essere risolti almeno una volta, allora un algoritmo bottom-up di programmazione dinamica è più efficiente di un fattore costante rispetto a un algoritmo ricorsivo top-down con memorizzazione. Il motivo è che non c'è alcun costo per la ricorsione e si ha un costo minore per il mantenimento della tabella. Inoltre, la struttura regolare degli accessi alla tabella di alcuni problemi può essere utilizzata da un algoritmo di programmazione dinamica per ridurre ulteriormente i vincoli di tempo e di spazio. Tuttavia, se nello spazio dei sottoproblemi esistono dei sottoproblemi che non necessitano mai di essere risolti, allora una soluzione ricorsiva con memorizzazione ha il notevole vantaggio di risolvere solo quei sottoproblemi che sono strettamente necessari.

Esercizi

16.2-1 Si confronti l'equazione di ricorrenza (16.4) con l'equazione di ricorrenza (8.4) che viene fuori dall'analisi del tempo medio di esecuzione dell'algoritmo del quicksort. Si dia la motivazione intuitiva del perché le soluzioni delle due equazioni di ricorrenza debbano essere così diverse.

16.2-2 Qual è il modo più efficiente per determinare il numero ottimo di moltiplicazioni scalari nel problema del prodotto di una sequenza di matrici: enumerare tutte le possibili parentesizzazioni del prodotto e calcolare, per ogni parentesizzazione, il numero di moltiplicazioni scalari, oppure eseguire la procedura RECURSIVE-MATRIX-CHAIN? Si giustifichi la risposta.

16.2-3 Si disegni l'albero di ricorsione della procedure MERGE-SORT, descritta nel paragrafo 1.3.1, quando viene applicata a un vettore con 16 elementi. Si spieghi il motivo per cui la tecnica della ricorsione con memorizzazione non produce un aumento di efficienza quando viene applicata a un buon algoritmo di tipo divide-et-impera come il MERGE-SORT.

16.3 Il problema della più lunga sottosequenza comune

Il prossimo problema che analizzeremo è il problema della più lunga sottosequenza comune. Una sottosequenza di una certa sequenza è la sequenza originale con l'esclusione di alcuni elementi (eventualmente nessuno). Formalmente, data una sequenza $X = \langle x_1, x_2, \ldots, x_m \rangle$, una sequenza $Z = \langle z_1, z_2, \ldots, z_k \rangle$ viene detta ***sottosequenza*** di X se esiste una sequenza, strettamente crescente, $\langle i_1, i_2, \ldots, i_k \rangle$ di indici di X tale che per tutti gli indici $j = 1, 2, \ldots, k$, vale

$x_{i_j} = z_j$. Per esempio, $Z = \langle B, C, D, B \rangle$ è una sottosequenza di $X = \langle A, B, C, B, D, A, B \rangle$, dove la corrispondente sequenza di indici risulta $\langle 2, 3, 5, 7 \rangle$.

Date due sottosequenze X e Y, diciamo che una sequenza Z è una ***sottosequenza comune*** di X e Y se Z è una sottosequenza di entrambe le sequenze X e Y. Per esempio, se $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$ la sequenza $\langle B, C, A \rangle$ è una sottosequenza comune di X e Y. La sequenza $\langle B, C, A \rangle$ non è la *più lunga* sottosequenza comune (LCS)[2] di X e Y, dato che è lunga 3 e la sequenza $\langle B, C, B, A \rangle$, che è comune a X e Y, ha lunghezza 4. La sequenza $\langle B, C, B, A \rangle$ è una LCS di X e Y, come pure la sequenza $\langle B, D, A, B \rangle$: non esiste alcuna sottosequenza comune di lunghezza maggiore o uguale a 5.

Nel ***problema della più lunga sottosequenza comune*** sono date due sequenze $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ e si vuole determinare la sottosequenza comune di X e Y di lunghezza massima. In questo paragrafo mostreremo come il problema della LCS possa essere risolto in modo efficiente con la tecnica della programmazione dinamica.

Caratterizzazione della più lunga sottosequenza comune

Un metodo di soluzione ingenuo per il problema della LCS consiste nell'enumerare tutte le sottosequenze di X e nel controllare che ogni sottosequenza sia anche una sottosequenza di Y. Durante ogni fase del procedimento risolutivo bisogna mantenere l'informazione sulla più lunga sottosequenza trovata fino a quel punto. Dato che ogni sottosequenza di X corrisponde a un sottoinsieme $\langle 1, 2, \ldots, m \rangle$ di indici di X, la soluzione ingenua richiede un tempo esponenziale, rendendola, pertanto, impraticabile per sequenze lunghe.

Tuttavia, il problema della LCS soddisfa la proprietà della sottostruttura ottima, come dimostrato dal teorema seguente. Come vedremo nel seguito, la classe naturale dei sottoproblemi del problema della LCS consiste nelle coppie di "prefissi" delle due sequenze di ingresso. Più precisamente, data una sequenza $X = \langle x_1, x_2, \ldots, x_m \rangle$ diciamo che X_i è il ***prefisso*** i-esimo della sequenza X se e solo se $X_i = \langle x_1, x_2, \ldots, x_i \rangle$, con $i = 1, 2, \ldots, m$. Per esempio, se $X = \langle A, B, C, B, D, A, B \rangle$, allora $X_4 = \langle A, B, C, B \rangle$ e X_0 è la sequenza vuota.

Teorema 16.1 (Sottostruttura ottima di una LCS)

Siano $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ due sequenze e sia $Z = \langle z_1, z_2, \ldots, z_k \rangle$ una qualunque LCS di X e Y.

1. Se $x_m = y_n$, allora $z_k = x_m = y_n$ e Z_{k-1} è una LCS di X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$ e $z_k \neq x_m$, allora Z è una LCS di X_{m-1} e Y.
3. Se $x_m \neq y_n$ e $z_k \neq y_n$, allora Z è una LCS di X e Y_{n-1}.

Dimostrazione. (1) Se fosse $z_k \neq x_m$, allora si potrebbe costruire una sottosequenza comune di X e Y di lunghezza $k + 1$: basterebbe concatenare a Z l'elemento $x_m = y_n$. Chiaramente, questo è una contraddizione dell'ipotesi che Z sia la *più lunga* sottosequenza comune di X e Y. Pertanto, deve necessariamente valere che $z_k = x_m = y_n$. Per le ipotesi fatte, il prefisso Z_{k-1} è una sottosequenza comune di X_{m-1} e Y_{n-1} di lunghezza $k - 1$. Dimostriamo che questo prefisso è una LCS. La dimostrazione è una dimostrazione per assurdo. Supponiamo che esista una sottosequenza comune W di X_{m-1} e Y_{n-1} di lunghezza maggiore di $k - 1$. Allora concatenando

[2] N.d.T.: LCS è l'acronimo del termine inglese Longest Common Subsequence.

W con l'elemento $x_m = y_n$ si ottiene una sottosequenza comune di X e Y di lunghezza maggiore di k. Pertanto otteniamo una contraddizione dell'ipotesi.

(2) Se $z_k \neq x_m$, allora Z è una sottosequenza comune di X_{m-1} e Y. Se esistesse una sottosequenza comune W di X_{m-1} e Y di lunghezza maggiore di k, allora W sarebbe anche una sottosequenza comune di X_m e Y. Questa è una contraddizione dell'ipotesi che Z sia una LCS di X e Y.

(3) Si dimostra con un ragionamento simmetrico a quello usato nel punto (2). ■

Il Teorema di caratterizzazione 16.1 dimostra che una LCS di due sequenze contiene al suo interno una LCS dei prefissi delle due sequenze. Quindi, il problema della più lunga sottosequenza comune soddisfa la proprietà della sottostruttura ottima. Inoltre, come vedremo nel seguito, una soluzione ricorsiva soddisfa anche la proprietà dei sottoproblemi comuni.

Soluzione ricorsiva dei sottoproblemi

Il Teorema 16.1 implica che, quando si vuole determinare una LCS di $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, si devono considerare uno o due sottoproblemi. Infatti, se $x_m = y_n$, per determinare una LCS di X e Y basta trovare una LCS di X_{m-1} e Y_{n-1} e concatenare a questa LCS l'elemento $x_m = y_n$. Nel caso in cui $x_m \neq y_n$, allora si devono risolvere due sottoproblemi: bisogna trovare sia una LCS di X_{m-1} e Y che una LCS di X e Y_{n-1}. La più grande di queste due LCS è una LCS di X e Y.

È facile notare che il problema della LCS soddisfa la proprietà dei sottoproblemi comuni. Per trovare una LCS di X e Y si deve necessariamente trovare sia una LCS di X_{m-1} e Y sia una LCS di X e Y_{n-1}. Entrambi i sottoproblemi devono risolvere il problema di trovare una LCS di X_{m-1} e Y_{n-1}. Chiaramente, molti altri sottoproblemi devono affrontare e risolvere i medesimi sottoproblemi.

Come nel caso del problema del prodotto di una sequenza di matrici, la nostra soluzione ricorsiva del problema della LCS comporta la definizione di una equazione di ricorrenza per determinare il costo di una soluzione ottima. Definiamo $c[i, j]$ come la lunghezza di una LCS delle sequenze X_i e Y_j. Se uno o l'altro degli indici i e j è uguale a 0, allora una delle due sottosequenze ha lunghezza 0, quindi la LCS ha lunghezza 0. La sottostruttura ottima del problema LCS permette di derivare la formula ricorsiva

$$c[i, j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 , \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j . \end{cases} \tag{16.5}$$

Calcolo della lunghezza di una LCS

Utilizzando l'equazione di ricorrenza (16.5) possiamo facilmente scrivere un algoritmo ricorsivo, esponenziale in tempo, che calcola la lunghezza di una LCS di due sequenze. Tuttavia, ci sono solamente $\Theta(mn)$ sottoproblemi distinti e quindi possiamo utilizzare la tecnica della programmazione dinamica per trovare le soluzioni con una modalità bottom-up.

La procedura LCS-LENGTH ha come parametri di ingresso le due sequenze $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, i valori $c[i, j]$ sono memorizzati in una tabella $c[0 \ldots m, 0 \ldots n]$ e gli elementi della tabella sono calcolati seguendo l'ordine delle righe. (In altri termini, prima si calcolano, da sinistra verso destra, gli elementi della prima riga, poi quelli della seconda riga e così via.) La procedura utilizza anche una seconda tabella $b[1 \ldots m, 1 \ldots n]$ per semplificare la costruzione di una soluzione ottima. L'idea intuitiva è che il generico elemento $b[i, j]$ "punta" alla posizione della tabella che corrisponde alla scelta della soluzione ottima del sottoproblema nel calcolo di $c[i, j]$. La procedura restituisce le due tabelle b e c: il valore dell'elemento $c[m, n]$ è la lunghezza di una LCS di X e Y.

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j] ← 0
for i ← 1 to m
    do for j ← 1 to n
          do if x_i = y_j
                then  c[i, j] ← c[i − 1, j − 1] + 1
                      b[i, j] ← "↖"
                else if c[i − 1, j] ≥ c[i, j − 1]
                        then  c[i, j] ← c[i − 1, j]
                              b[i, j] ← "↑"
                        else  c[i, j] ← c[i, j − 1]
                              b[i, j] ← "←"
return b, c
```

La figura 16.3 mostra le tabelle restituite come risultato dalla chiamata della procedura LCS-LENGTH con le sequenze $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$. Ogni elemento della tabella è calcolato in tempo $O(1)$ e quindi il tempo di esecuzione della procedura è $O(mn)$.

Costruzione di una LCS

La tabella b restituita dalla procedura LCS-LENGTH può essere utilizzata per costruire direttamente una LCS per le sequenze $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$. L'idea è piuttosto semplice. Consideriamo l'elemento $b[m, n]$ e poi percorriamo la tabella nel senso indicato dai puntatori (le frecce). Quando il valore dell'elemento $b[i, j]$ è uguale a "↖", significa che $x_i = y_j$ appartiene alla LCS. Utilizzando questa strategia gli elementi della LCS vengono scanditi in ordine inverso. La seguente procedura ricorsiva stampa una LCS di X e Y nel giusto ordine. La chiamata iniziale della procedura è PRINT-LCS(b, X, *length*[X], *length*[Y]).

i \ j		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Figura 16.3 *Le tabelle c e b calcolate dalla procedura* LCS-LENGTH *applicata alle sequenze* $X = \langle A, B, C, B, D, A, B \rangle$ *e* $Y = \langle B, D, C, A, B, A \rangle$. *L'elemento della tabella di riga i e colonna j contiene il valore* $c[i, j]$ *e la freccia che rappresenta il valore di* $b[i, j]$. *Il valore* 4 *dell'elemento* $c[7, 6]$ – *nell'angolo inferiore a destra nella tabella – è la lunghezza di una LCS* $\langle B, C, B, A \rangle$ *di X e Y. Per valori degli indici* $i, j > 0$, *il valore dell'elemento* $c[i, j]$ *dipende solamente dal fatto che* x_i *sia uguale a* y_j *e dai valori degli elementi* $c[i-1, j]$, $c[i, j-1]$ *e* $c[i-1, j-1]$ *che sono calcolati prima di* $c[i, j]$. *Per ricostruire gli elementi di una LCS è sufficiente seguire le frecce* $b[i, j]$ *a partire dall'angolo inferiore destro della tabella. Ogni freccia del tipo "↖" sul cammino corrisponde a un elemento (in chiaro nella figura) che appartiene alla LCS per cui vale* $x_i = y_j$.

```
PRINT-LCS(b, X, i, j)
if i = 0 oppure j = 0
    then return
if b[i, j] = "↖"
    then PRINT-LCS(b, X, i − 1, j − 1)
         print x_i
    else if b[i, j] = "↑"
             then PRINT-LCS (b, X, i − 1, j)
             else PRINT-LCS(b, X, i, j − 1)
```

Per esempio, considerando la tabella b descritta nella figura 16.3, la procedura precedente stampa "*BCBA*". Dato che in tutti i livelli della ricorsione si decrementa sempre uno degli indici i e j, si ha che il tempo di esecuzione della procedura è $O(m + n)$.

Una soluzione più efficiente

Quando si è concluso il progetto di un algoritmo, capita spesso di accorgersi che il tempo di esecuzione e la quantità di memoria utilizzata dall'algoritmo possono essere migliorati. Questo accade molto frequentemente nel caso di algoritmi che hanno una soluzione immediata con tecniche di programmazione dinamica.

Alcune modifiche possono semplificare la struttura dell'algoritmo e migliorare l'efficienza di un fattore costante, ma senza portare a miglioramenti asintotici della efficienza. In alcuni casi, tuttavia, le modifiche possono portare a sostanziali miglioramenti asintotici sia in tempo che in spazio.

Per esempio, la soluzione del problema della LCS potrebbe non utilizzare la tabella b. Infatti, il valore di un generico elemento $c[i, j]$ dipende solamente dal valore di altri tre elementi della tabella c: $c[i-1, j-1]$, $c[i-1, j]$ e $c[i, j-1]$. Dato il valore di $c[i, j]$ possiamo determinare, in tempo $O(1)$ e senza accedere alla tabella b, quale di questi tre valori è stato utilizzato per calcolare $c[i, j]$. Quindi, utilizzando una procedura molto simile alla procedura PRINT-LCS possiamo costruire una LCS in tempo $O(m + n)$. (Nell'Esercizio 16.3-2 si chiede di scrivere la procedura.) Sebbene questa modifica riduca lo spazio richiesto di una quantità $\Theta(mn)$, lo spazio necessario per il calcolo di una LCS non diminuisce in modo asintotico: infatti abbiamo sempre bisogno di uno spazio dell'ordine $\Theta(mn)$ per memorizzare la tabella c.

Tuttavia, possiamo diminuire lo spazio asintotico richiesto dalla procedura LCS-LENGTH: infatti la procedura, a ogni istante, necessita di sole due righe della tabella c: la riga calcolata in precedenza e la riga in esame. (In realtà, si può utilizzare una quantità di spazio solo leggermente maggiore dello spazio richiesto per memorizzare *una* riga della tabella c, si veda l'Esercizio 16.3-4.) Questi miglioramenti sono effettivi solo nel caso in cui si debba determinare la lunghezza di una LCS; se il problema richiede di restituire gli elementi di una LCS, allora la tabella di dimensioni minori non conterrebbe abbastanza informazioni per ricostruire una LCS in un tempo dell'ordine di $O(m + n)$.

Esercizi

16.3-1 Si determini una LCS di $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ e $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

16.3-2 Si mostri come sia possibile ricostruire una LCS in tempo $O(m + n)$ senza utilizzare la tabella b, ma avendo a disposizione la tabella c e le sequenze originali $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$.

16.3-3 Si dia una versione ricorsiva con memorizzazione della procedura LCS-LENGTH eseguibile in tempo $O(mn)$.

16.3-4 Si mostri come calcolare la lunghezza di una LCS utilizzando solamente $2 \min(m, n)$ elementi della tabella c più uno spazio addizionale $O(1)$. Successivamente, si mostri come risolvere lo stesso problema con solo $\min(m, n)$ elementi più uno spazio addizionale $O(1)$.

16.3-5 Si determini un algoritmo per trovare in tempo $O(n^2)$ la più lunga sottosequenza monotona (crescente) di una sequenza di n numeri.

* *16.3-6* Si determini un algoritmo per trovare in tempo $O(n \lg n)$ la più lunga sottosequenza monotona (crescente) di una sequenza di n numeri. (*Suggerimento*: si noti che elemento di una sottosequenza candidata di lunghezza i è grande almeno quanto l'ultimo elemento di una sottosequenza candidata di lunghezza $i - 1$. Si memorizzino le sottosequenze candidate collegandole assieme nella sequenza d'ingresso.)

16.4 Triangolazione ottima di un poligono

In questo paragrafo esamineremo il problema della triangolazione ottima di un poligono convesso. A dispetto della apparenza esteriore, vedremo che questo problema di natura geometrica presenta delle considerevoli e sorprendenti analogie con il problema del prodotto di una sequenza di matrici.

Un ***poligono*** è una spezzata chiusa del piano bidimensionale. Ovvero, un poligono è una spezzata che si richiude su se stessa ed è formata da una sequenza di segmenti di retta, detti i ***lati*** del poligono. Un punto che unisce due lati consecutivi è chiamato un ***vertice*** del poligono. Un poligono è detto ***semplice*** se non contiene intersezioni con se stesso; in questo paragrafo considereremo sempre poligoni semplici. L'insieme dei punti del piano racchiusi da un poligono semplice è detto ***interno*** del poligono. L'insieme dei punti del piano che stanno sui lati del poligono è detto ***perimetro***. Infine, l'insieme dei punti che stanno all'esterno del poligono è detto ***esterno***. Un poligono semplice è detto ***convesso*** se, presi due punti qualsiasi che appartengono al perimetro, o all'interno del poligono, il segmento di retta che li unisce è formato da punti che appartengono al perimetro o all'interno del poligono.

Generalmente un poligono viene rappresentato elencando, in senso orario, l'insieme dei suoi vertici. Pertanto, se $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ è un poligono convesso, allora P è costituito da n lati $\overline{v_0 v_1}, \overline{v_1 v_2}, \ldots, \overline{v_{n-1} v_n}$, dove v_n viene interpretato come v_0. (Nel seguito tutte le operazioni aritmetiche che coinvolgono gli indici dei vertici del poligono sono sempre definite modulo il numero dei vertici del poligono.)

Siano v_i e v_j due vertici non adiacenti, il segmento $\overline{v_i v_j}$ è detto ***corda*** del poligono: una corda $\overline{v_i v_j}$ divide un poligono in due poligoni $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ e $\langle v_j, v_{j+1}, \ldots, v_i \rangle$. Una ***triangolazione*** di un poligono consiste in un insieme T di corde del poligono che divide il poligono in un insieme di triangoli (poligoni con tre lati) disgiunti. La figura 16.4 descrive due possibili triangolazioni di un poligono di 7 lati. In una triangolazione le corde non hanno intersezioni (con la sola eccezione dei punti terminali), inoltre l'insieme T delle corde è massimale: una corda che non appartiene all'insieme T interseca almeno una delle corde in T. I lati dei triangoli risultanti da una triangolazione possono essere sia corde della triangolazione che lati del poligono. Infine, ogni triangolazione di un poligono convesso con n vertici ha $n - 3$ corde e divide il poligono in $n - 2$ triangoli.

Nel ***problema della triangolazione ottima (di un poligono)*** consideriamo un poligono convesso $P = \langle v_0, v_1, \ldots, v_{n-1} \rangle$ ed una funzione peso w definita sui triangoli formati dai lati e dalle corde di P. Il problema consiste nel determinare una triangolazione che minimizza la somma dei pesi dei triangoli nella triangolazione del poligono. Una funzione peso associata in modo naturale ai triangoli è

$$w(\Delta v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i| ,$$

dove $|v_i v_j|$ è la distanza euclidea tra v_i e v_j. L'algoritmo che stiamo progettando non dipende dalla scelta della funzione peso: la funzione peso può essere definita in modo arbitrario.

Figura 16.4 *Due triangolazioni di un poligono convesso. Ogni triangolazione di un poligono con 7 lati ha un numero di corde uguale a* $7 - 3 = 4$ *e divide il poligono in un numero di triangoli pari a* $7 - 2 = 5$.

Analogia con la parentesizzazione

Il problema della triangolazione di un poligono presenta delle sorprendenti analogie con il problema della parentesizzazione di una espressione come quella che descrive il prodotto di una sequenza di matrici. Il modo migliore per spiegare questa analogia è fare riferimento a strutture ad albero.

Una parentesizzazione completa di una espressione corrisponde a un albero binario completo, altrimenti detto ***albero di parsing*** dell'espressione. La figura 16.5(a) descrive l'albero di parsing della parentesizzazione del prodotto della sequenza di matrici

$$((A_1(A_2A_3))(A_4(A_5A_6))) . \tag{16.6}$$

Ogni foglia dell'albero di parsing è etichettata con uno dei componenti atomici (matrici) dell'espressione. Un sottoalbero di un albero di parsing rappresenta l'espressione $(E_i E_j)$, se la radice del sottoalbero ha un sottoalbero sinistro, che rappresenta l'espressione E_i, ed un sottoalbero destro, che rappresenta l'espressione E_j. Esiste una corrispondenza biunivoca tra gli alberi di parsing e le espressioni di n elementi atomici parentesizzate completamente.

Una triangolazione di un poligono convesso $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ può essere anch'essa rappresentata da un albero di parsing. La figura 16.5(b) descrive l'albero di parsing che corrisponde alla triangolazione del poligono della figura 16.4(a). I nodi interni dell'albero di parsing sono le corde della triangolazione, mentre la radice dell'albero è il lato $\overline{v_0 v_n}$; le foglie dell'albero sono i rimanenti lati del poligono. La radice $\overline{v_0 v_n}$ è un lato del triangolo $\Delta v_0 v_k v_n$; è questo triangolo che determina quali siano i figli della radice: uno è la corda $\overline{v_0 v_k}$, l'altro è la corda $\overline{v_k v_n}$. Si noti come questo triangolo divida il poligono originale in tre parti: il triangolo $\Delta v_0 v_k v_n$, il poligono $\langle v_0, v_1, \ldots, v_k \rangle$, ed il poligono $\langle v_k, v_{k+1}, \ldots, v_n \rangle$. Inoltre, i lati dei due sottopoligoni sono costituiti da lati del poligono originale, con l'eccezione delle loro radici, che sono corde $\overline{v_0 v_k}$ e $\overline{v_k v_n}$. Ragionando in modo ricorsivo possiamo dire che il poligono $\langle v_0, v_1, \ldots, v_k \rangle$ contiene il sottoalbero di sinistra dell'albero di parsing, mentre il poligono $\langle v_k, v_{k+1}, \ldots, v_n \rangle$ contiene il sottoalbero di destra.

Figura 16.5 *Alberi di parsing.* (a) *L'albero di parsing del prodotto di matrici* $((A_1(A_2A_3))(A_4(A_5A_6)))$ *e della triangolazione del poligono con 7 lati della figura 16.4(a).* (b) *La triangolazione del poligono sovrapposta all'albero di parsing. Ogni matrice* A_i *corrisponde al lato* $\overline{v_{i-1}v_i}$, *per* $i = 1, 2, \ldots, 6$.

In generale, quindi, una triangolazione di un poligono con n lati corrisponde a un albero di parsing con $n - 1$ foglie. Da un albero di parsing si può costruire, con un procedimento simmetrico, una triangolazione. Infatti, gli alberi di parsing sono in corrispondenza biunivoca con le triangolazioni di poligoni.

Il prodotto completamente parentesizzato di n matrici corrisponde a un albero di parsing con n foglie, e, pertanto, corrisponde anche a una triangolazione di un poligono con $n + 1$ vertici. Le figure 16.5(a) e (b) mettono in evidenza questa corrispondenza biunivoca: ogni matrice A_i del prodotto $A_1 A_2 \cdots A_n$ corrisponde a un lato $\overline{v_{i-1}v_i}$ di un poligono con $n + 1$ vertici, mentre una corda $\overline{v_iv_j}$ con $i < j$, corrisponde alla matrice $A_{i+1..j}$ calcolata durante la valutazione del prodotto matriciale.

Difatti, il problema del prodotto di una sequenza di matrici è un caso particolare del problema della triangolazione ottima. In altri termini, ogni istanza del problema del prodotto di una sequenza di matrici può essere, equivalentemente, riformulata come un problema di triangolazione ottima. Dato il prodotto di una sequenza di matrici $A_1 A_2 \ldots A_n$, definiamo un poligono convesso $P = \langle v_0, v_1, \ldots, v_n \rangle$ con $n + 1$ vertici. Se la matrice A_i ha dimensioni $p_{i-1} \times p_i$, con $i = 1, 2, \ldots, n$, allora la funzione peso è definita

$$w(\Delta v_i v_j v_k) = p_i p_j p_k.$$

Una triangolazione ottima del poligono P rispetto a questa funzione peso definisce l'albero di parsing della parentesizzazione ottima del prodotto $A_1 A_2 \ldots A_n$.

Sebbene la proprietà simmetrica non sia vera – il problema della triangolazione ottima *non* è un caso particolare del problema del prodotto di una sequenza di matrici – accade che la procedura MATRIX-CHAIN-ORDER, definita nel paragrafo 16.1, con alcune piccole modifiche sia in grado di risolvere il problema della triangolazione di un poligono con $n + 1$ vertici. L'idea di base è di sostituire la sequenza $\langle p_0, p_1, \ldots, p_n \rangle$ delle dimensioni delle matrici con la sequenza $\langle v_0, v_1, \ldots, v_n \rangle$ dei vertici del poligono. Chiaramente ogni riferimento a p deve diventare un riferimento a v, ed inoltre la linea 9 viene così modificata:

9 do $q \leftarrow m[i, k] + m[k + 1, j] + w(\Delta v_{i-1} v_k v_j)$

Dopo avere eseguito l'algoritmo, l'elemento $m[1, n]$ conterrà il peso della triangolazione ottima. Nel seguito, descriviamo le ragioni per cui vale questa proprietà.

Sottostruttura di una triangolazione ottima

Sia T una triangolazione ottima di un poligono convesso $P = \langle v_0, v_1, \ldots, v_n \rangle$ con $n + 1$ vertici che contenga il triangolo $\Delta v_0 v_k v_n$, per un qualche k, dove $1 \le k \le n - 1$. Il peso di T è dato dalla somma dei pesi di $\Delta v_0 v_k v_n$ e dei triangoli che appartengono alla triangolazione dei due sottopoligoni $\langle v_0, v_1, \ldots, v_k \rangle$ e $\langle v_k, v_{k+1}, \ldots, v_n \rangle$. La triangolazione dei sottopoligoni definiti da T deve essere necessariamente ottima, poiché una triangolazione di uno qualunque dei due sottopoligoni con un peso minore contraddirebbe l'ipotesi di minimalità del peso di T.

Soluzione ricorsiva

Allo stesso modo in cui abbiamo definito $m[i, j]$ come il costo minimo del calcolo del prodotto matriciale $A_i A_{i+1} \cdots A_j$, definiamo $t[i, j]$, per $1 \le i < j \le n$, come il peso di una triangolazione ottima del poligono $\langle v_{i-1}, v_i, \ldots, v_j \rangle$. Per facilitare il ragionamento, assegneremo peso 0 a ogni poligono degenere del tipo $\langle v_{i-1}, v_i \rangle$. Il peso di una triangolazione ottima del poligono P è, quindi, dato dal valore dell'elemento $t[1, n]$.

A questo punto cerchiamo di definire $t[i, j]$ in modo ricorsivo. La base della ricorsione è il caso degenere di un poligono con due soli vertici: $t[i, i] = 0$, per $i = 1, 2, \ldots, n$. Quando $j - i \ge 1$, la sequenza di vertici $\langle v_{i-1}, v_i, \ldots, v_j \rangle$ definisce un poligono con almeno tre vertici. Il nostro obiettivo è quello di minimizzare, per tutti i vertici v_k, con $k = i, i + 1, \ldots, j - 1$, la somma tra il peso di $\Delta v_{i-1} v_k v_j$ ed i pesi delle triangolazioni ottime dei poligoni $\langle v_{i-1}, v_i, \ldots, v_k \rangle$ e $\langle v_k, v_{k+1}, \ldots, v_j \rangle$. La formulazione ricorsiva risulta:

$$t[i, j] = \begin{cases} 0 & \text{se } i = j, \\ \min\limits_{i \le k \le j-1} \{t[i, k] + t[k + 1, j] + w(\Delta v_{i-1} v_k v_j)\} & \text{se } i < j. \end{cases} \tag{16.7}$$

Si confronti la precedente equazione di ricorrenza con l'equazione di ricorrenza (16.2) che è stata definita per il calcolo del minimo numero di moltiplicazioni scalari $m[i, j]$ necessarie per calcolare il prodotto matriciale $A_i A_{i+1} \cdots A_j$. Se si eccettua la presenza della funzione peso, le due equazioni di ricorrenza sono sostanzialmente identiche; pertanto, con le piccole modifiche discusse in precedenza, la procedura MATRIX-CHAIN-ORDER è in grado di calcolare il peso di una triangolazione ottima. Allo stesso modo della procedura MATRIX-CHAIN-ORDER, la procedura della triangolazione ottima viene eseguita in un tempo $\Theta(n^3)$ con un uso di spazio $\Theta(n^2)$.

Esercizi

16.4-1 Si dimostri che ogni triangolazione di un poligono convesso con n vertici ha $n - 3$ corde e divide il poligono in $n - 2$ triangoli.

16.4-2 Il professor Guinevere sostiene che un algoritmo più veloce per risolvere il problema della triangolazione ottima si possa ottenere considerando come funzione peso la misura dell'area di un triangolo. L'opinione del professore è corretta?

16.4-3 Si supponga che la funzione peso dipenda dalle corde di una triangolazione invece che dai triangoli. Il peso della triangolazione rispetto a questa funzione peso w è dato dalla somma dei pesi delle corde della triangolazione. Si dimostri che il problema della triangolazione con corde pesate è un caso particolare del problema della triangolazione con triangoli pesati.

16.4-4 Si determini una triangolazione ottima di un ottagono regolare con lati di lunghezza unitaria. Si utilizzi la seguente funzione peso

$$w(\Delta v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i| ,$$

dove $|v_i v_j|$ è la distanza euclidea tra v_i e v_j. (Un poligono è detto regolare se tutti i lati e tutti gli angoli interni sono uguali.)

Problemi

16-1 Il problema del commesso viaggiatore bitonico e euclideo

Il problema del commesso viaggiatore euclideo richiede di determinare il più breve cammino chiuso che connette un insieme di n punti del piano. Nella sua formulazione generale il problema è NP-completo, e pertanto si ritiene che non esistano soluzioni polinomiali nel tempo. (Cfr. Capitolo 36.)

J. L. Bentley ritiene che il problema possa essere notevolmente semplificato se si considerano solamente i ***cammini bitonici***. I cammini bitonici sono quei cammini che partono dal punto più a sinistra e vanno da sinistra verso destra finché non viene raggiunto il punto più a destra, e poi tornano al punto di partenza collegando punti da destra verso sinistra. La figura 16.6(b) descrive il più breve cammino bitonico su un insieme di 7 punti. Nel caso dei cammini bitonici è possibile definire un algoritmo di risoluzione che risulta polinomiale nel tempo.

Si progetti un algoritmo che determina un cammino bitonico ottimo in tempo $O(n^2)$. Si assuma che non esistano punti del piano con la stessa ascissa. (*Suggerimento*: si effettui una scansione da sinistra verso destra mantenendo le possibilità ottime per le due parti del cammino.)

16-2 La stampa precisa

Si consideri il problema della stampa precisa di un paragrafo su una stampante. Il testo in ingresso è una sequenza di n parole di lunghezza $l_1, l_2, \ldots, l_n$, dove la lunghezza di una parola è data dal numero dei caratteri che compongono la parola. Si vuole stampare questo paragrafo in un certo numero di linee, con il vincolo che ogni linea può contenere al massimo M caratteri. Il criterio adottato per dire che una stampa è "precisa" è il seguente. Assumiamo che una certa linea contenga tutte le parole che vanno dalla parola i alla parola j, con $i \leq j$, e che due parole consecutive siano separate esattamente da uno spazio: allora il numero di spazi necessari per completare la riga è dato da $M - j + i - \sum_{k=i}^{j} l_k$ e questo numero non deve essere negativo.

Figura 16.6 *Sette punti del piano su una griglia con quadrati di dimensione unitaria.* (a) *Il più corto cammino chiuso di lunghezza* 24.88.... *Questo cammino non è bitonico.* (b) *Il più corto cammino bitonico per lo stesso insieme di punti. La sua lunghezza è* 25.58....

Si vuole minimizzare la somma, relativamente a tutte le righe con l'eccezione dell'ultima, del cubo del numero di spazi ausiliari necessari per completare ogni riga. Si determini un algoritmo di programmazione dinamica che stampi in modo preciso un paragrafo di n righe su una stampante. Si analizzino sia il tempo di esecuzione che i vincoli di spazio dell'algoritmo proposto.

16-3 Distanza di editing

Quando un terminale "intelligente" modifica una linea di testo, sostituisce una esistente stringa "sorgente" $x[1 \ldots m]$ con una stringa "risultato" $y[1 \ldots n]$. Questa modifica può essere fatta in modi diversi. Un generico carattere della stringa sorgente può essere cancellato, sostituito da un altro carattere, oppure copiato nella stringa risultato; altri caratteri possono essere inseriti, oppure due caratteri adiacenti della stringa sorgente possono essere scambiati nel momento in cui sono copiati nella stringa risultato. Dopo avere eseguito un certo numero di operazioni, può accadere di cancellare un intero *suffisso* della stringa sorgente.

Per esempio, supponiamo di volere trasformare la stringa sorgente "algoritmo" nella stringa risultato "altrove". Un modo per effettuare questa trasformazione è definito dalla seguente lista di operazioni.

Operazione	Stringa risultato	Stringa sorgente
copiare a	a	lgoritmo
copiare l	al	goritmo
sostituire g con t	alt	oritmo
scambiare or con ro	altro	itmo
inserire v	altrov	itmo
inserire e	altrove	itmo
cancellare itmo	altrove	

Chiaramente, ci sono altre sequenze di operazioni che producono lo stesso effetto.

Le operazioni cancellare, sostituire, copiare, inserire, scambiare e cancellare un suffisso hanno un certo costo. (Ci si aspetta che il costo dell'operazione che sostituisce un carattere sia minore della somma dei costi delle operazioni che cancellano ed inseriscono un carattere. Nel caso contrario, l'operazione di sostituzione non dovrebbe essere utilizzata.) Il costo di una certa sequenza di operazioni di modifica è dato dalla somma del costo delle singole operazioni nella sequenza. Nel nostro esempio, il costo della trasformazione della parola "algoritmo" nella parola "altrove" risulta

2 costo(copiare) + costo(sostituire) + costo(scambiare) + 2 costo(inserire) + costo(suffisso).

Si considerino due sequenze $x[1 \,..\, m]$ e $y[1 \,..\, n]$ ed un insieme di costi di operazioni: la ***distanza di editing*** tra x e y è definita come il costo della sequenza più economica di operazioni che trasforma x in y. Si vuole definire un algoritmo di programmazione dinamica che determina la distanza di editing tra $x[1 \,..\, m]$ e $y[1 \,..\, n]$ e stampa la sequenza ottima delle trasformazioni. Si analizzino sia il tempo di esecuzione che i vincoli di spazio dell'algoritmo proposto.

16-4 La pianificazione di una festa aziendale

Il presidente dell'azienda A.-B. Corporation ha chiesto al professor McKenzie di effettuare una consulenza per risolvere il problema della pianificazione di una festa aziendale. L'azienda ha una struttura gerarchica: più precisamente, la relazione "essere supervisore di" definisce un albero la cui radice è il presidente dell'azienda. L'ufficio del personale ha quantificato con un numero reale la "convivialità" di ogni impiegato. Affinché la festa aziendale risulti piacevole per tutti i partecipanti, il presidente non desidera che alla festa partecipino sia un impiegato che il suo immediato supervisore.

a. Si definisca un algoritmo per preparare la lista degli invitati. L'obiettivo è di massimizzare la somma della convivialità degli ospiti. Si analizzi il tempo di esecuzione proposto.

b. Come può il professore esser certo che il presidente sia invitato alla festa aziendale?

16-5 L'algoritmo di Viterbi

Le tecniche della programmazione dinamica possono essere anche utilizzate nel caso di algoritmi per il riconoscimento del parlato, basati su particolari grafi orientati $G = (V, E)$. Ogni arco $(u, v) \in E$ è etichettato con un suono $\sigma(u, v)$ appartenente a un insieme finito Σ di suoni. Questi grafi etichettati definiscono un modello formale per l'analisi del parlato in un linguaggio particolarmente semplice. Ogni cammino nel grafo che parte da un determinato vertice v_0 corrisponde a una possibile sequenza di suoni. L'etichetta di un cammino orientato è definita dalla concatenazione delle etichette degli archi che appartengono al cammino.

a. Dato un grafo etichettato G, un vertice v_0 e una sequenza $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ di elementi di Σ, si definisca un algoritmo efficiente per trovare un cammino di G il cui vertice iniziale sia v_0 e la cui etichetta sia la sequenza s, se tale cammino esiste. Nel caso invece in cui il cammino non esista, l'algoritmo dovrà stampare il messaggio NESSUN CAMMINO. Si analizzi il tempo di esecuzione dell'algoritmo proposto. (*Suggerimento*: potrebbe essere utile fare riferimento a concetti introdotti nel Capitolo 23.)

Supponiamo di assegnare a ogni arco $(u, v) \in E$ una probabilità $p(u, v)$: il valore $p(u, v)$ indica la probabilità di attraversare l'arco (u, v) partendo da u ed emettendo il suono corrispondente. Naturalmente la somma delle probabilità degli archi che escono da un determinato vertice è 1. La probabilità di un cammino è definita come il prodotto delle probabilità degli archi che appartengono al cammino. La probabilità di un cammino, la cui origine è il vertice v_0, può essere interpretata come la probabilità che un attraversamento casuale del grafo a partire dal vertice v_0 seguirà quel cammino. Chiaramente la scelta di quale arco percorrere va fatta in accordo alle probabilità degli archi che escono dal vertice in esame.

b. Si modifichi l'algoritmo di risoluzione della parte (a) in modo tale che, se l'algoritmo restituisce un cammino, allora questo sia il *cammino più probabile* che parte da v_0 ed abbia s come etichetta. Si analizzi il tempo di esecuzione dell'algoritmo proposto.

Note al capitolo

Lo studio sistematico della programmazione dinamica è iniziato nel 1955 con i lavori di R. Bellman. In questo contesto, e anche nel caso della programmazione lineare, la parola "programmazione" si riferisce al fatto che i metodi di risoluzione sono basati su tabelle. Sebbene elementi di programmazione dinamica fossero già presenti in tecniche di ottimizzazione note in precedenza, si deve a Bellman la definizione della teoria matematica della programmazione dinamica [21].

Hu e Shing [106] hanno presentato un algoritmo che risolve il problema del prodotto di una sequenza di matrici con un tempo di esecuzione $O(n \lg n)$. Inoltre, hanno dimostrato la corrispondenza tra il problema della triangolazione ottima di un poligono ed il problema del prodotto di una sequenza di matrici.

La definizione di un algoritmo che risolva il problema della più lunga sottosequenza comune in tempo $O(mn)$ sembra non essere attribuibile a nessuno in particolare. Knuth [43] si è domandato se esistesse una soluzione non quadratica al problema della LCS. Una risposta positiva a questa domanda è stata data da Masek e Paterson [143] che hanno definito un algoritmo che, se le sequenze sono tratte da un insieme di dimensione limitata, viene eseguito in tempo $O(mn / \lg n)$, con $n \leq m$. Nel caso particolare in cui nessun elemento compaia più di una volta nella sequenza in ingresso, Szymansk [184] ha dimostrato che il problema può essere risolto in tempo $O((n + m) \lg (n + m))$. La maggior parte di questi risultati possono essere estesi al problema della distanza di editing (Problema 16-3).

Algoritmi greedy

17

Generalmente gli algoritmi per la risoluzione di problemi di ottimizzazione sono costituiti da sequenze di passi elementari in cui, a ogni passo, si presentano un certo numero di scelte alternative. Per alcuni problemi di ottimizzazione, l'uso di tecniche di programmazione dinamica per decidere la scelta migliore risulta inutilmente oneroso: lo stesso risultato potrebbe essere ottenuto con algoritmi più semplici e più efficienti. Gli ***algoritmi greedy***[1] adottano la strategia di prendere quella decisione che, al momento, appare come la migliore. In altri termini, viene sempre presa quella decisione che, localmente, è la decisione ottima, senza preoccuparsi se tale decisione porterà a una soluzione ottima del problema nella sua globalità. In questo capitolo verranno discussi quei problemi di ottimizzazione che possono essere risolti tramite gli algoritmi greedy.

In generale, non sempre esiste un algoritmo greedy che determina la soluzione ottima di un qualsiasi problema, tuttavia gli algoritmi greedy possono essere utilizzati per ottenere le soluzioni di alcune classi di problemi. Nel paragrafo 17.1 prenderemo in esame un problema non banale ma allo stesso tempo piuttosto semplice: il problema della selezione di attività. Questo problema può essere risolto in modo efficiente con un algoritmo greedy. Nel paragrafo 17.2, invece, presenteremo le caratteristiche di base del metodo greedy. Il paragrafo 17.3 presenta una applicazione molto importante delle tecniche greedy: il progetto di codici per la compressione dei dati (codici di Huffman). Nel paragrafo 17.4 esamineremo una parte della teoria che sta alla base di quelle strutture combinatorie dette "matroidi"; per queste strutture gli algoritmi greedy riescono sempre a ottenere una soluzione ottima. Infine, il paragrafo 17.5 presenta una applicazione dei matroidi alla risoluzione del problema della definizione della sequenza di esecuzione (scheduling) di programmi di durata unitaria con scadenze e penalità.

La tecnica greedy è potente e si adatta particolarmente bene a risolvere un'ampia classe di problemi. Nei capitoli successivi presenteremo un certo numero di algoritmi che possono essere visti come esempi di applicazione del metodo greedy. Tra questi ricordiamo gli algoritmi per trovare il minimo albero di copertura (Capitolo 24), l'algoritmo di Dijkstra dei cammini minimi da un'unica sorgente (Capitolo 25), l'algoritmo euristico di Chvátal per la copertura di un insieme (Capitolo 37). L'algoritmo per trovare il minimo albero di copertura è forse l'esempio classico di applicazione del metodo greedy. Sebbene non vi siano dipendenze logiche tra questo capitolo ed il Capitolo 24, potrebbe essere utile leggere i due capitoli contemporaneamente.

[1] N.d.T.: La traduzione letterale del termine inglese "greedy" è "ingordo" o "goloso".

17.1 Selezione di attività

Il nostro primo esempio consiste nel problema dell'assegnamento di una risorsa condivisa da un certo numero di attività in competizione tra loro. Dimostreremo che la selezione di un insieme che contiene il massimo numero di attività in competizione e mutuamente compatibili può essere ottenuta con un elegante ed efficiente algoritmo greedy.

Sia $S = \{1, 2, \ldots, n\}$ un insieme di n attività che intendono utilizzare una determinata risorsa, per esempio una sala di lettura, che non può essere utilizzata contemporaneamente da due o più attività. Una generica attività i è caratterizzata da un ***tempo di inizio*** (attivazione) s_i e da un ***tempo di fine*** (conclusione) f_i, dove $s_i \leq f_i$. Una volta selezionata, l'attività i è completata nell'intervallo di tempo (aperto a destra) $[s_i, f_i)$. Due attività i e j vengono dette ***compatibili*** se gli intervalli $[s_i, f_i)$ e $[s_j, f_j)$ non si sovrappongono (ovvero, i e j sono compatibili se vale che $s_i \geq f_j$ oppure $s_j \geq f_i$). Il ***problema della selezione di attività*** consiste nell'individuazione di un insieme che contiene il massimo numero di attività mutuamente compatibili.

Un algoritmo greedy che risolve il problema della selezione di attività è descritto dalla procedura seguente. Si assume che le attività in ingresso siano ordinate in modo crescente rispetto al loro tempo di fine:

$$f_1 \leq f_2 \leq \ldots \leq f_n \,. \tag{17.1}$$

Chiaramente, questa non è una limitazione, dato che possiamo sempre ordinare le attività in questo modo in tempo $O(n \lg n)$. La procedura assume che i parametri s e f siano dei vettori (array).

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n ← length[s]
2  A ← {1}
3  j ← 1
4  for i ← 2 to n
5      do if s_i ≥ f_j
6            then A ← A ∪ {i}
7                 j ← i
8  return A
```

Le operazioni eseguite dall'algoritmo sono visualizzate nella figura 17.1. L'insieme A contiene le attività selezionate. La variabile j indica l'ultima attività inserita nell'insieme A. Dato che le attività sono considerate in ordine non decrescente, rispetto al loro tempo di fine, abbiamo che f_j rappresenta sempre il massimo tempo di fine di tutte le attività che appartengono a A. Più precisamente,

$$f_j = \max\{f_k : k \in A\} \,. \tag{17.2}$$

Le linee 2-3 hanno il compito di selezionare l'attività numero 1: inizialmente l'insieme A contiene solamente questa attività e, pertanto, alla variabile j viene assegnato il valore 1. Le linee 4-7 considerano una dopo l'altra tutte le attività: una generica attività i viene inserita in A solo nel caso in cui sia compatibile con le altre attività selezionate in precedenza. Per vedere se una certa attività i è compatibile con tutte le attività selezionate fino a quel punto è sufficiente, per l'equazione (17.2), controllare (linea 5) che il tempo di inizio dell'attività s_i

Figura 17.1 *Le operazioni della procedura* GREEDY-ACTIVITY-SELECTOR *su di un insieme con undici attività. Le attività sono riportate nella parte sinistra della figura. Ogni riga della figura corrisponde a una iterazione del ciclo* for *(linee 4-7). Le attività che sono state selezionate in A sono rappresentate da rettangoli in grigio, mentre l'attività in esame è rappresentata da un rettangolo bianco. Se il tempo di inizio s_i dell'attività i cade prima del tempo di fine f_j della attività selezionata più recentemente (la freccia tra queste due attività punta a sinistra), allora l'attività non va selezionata. Altrimenti (la freccia è diretta in alto o a destra) l'attività va selezionata ed inserita nell'insieme A.*

non sia minore del tempo di fine f_j dell'attività inserita più recentemente in A. Se l'attività i risulta compatibile allora la si inserisce in A e si modifica il valore di j (linee 6-7). La procedura GREEDY-ACTIVITY-SELECTOR è piuttosto efficiente. Infatti, supponendo che tutte le attività

siano ordinate in modo crescente rispetto al loro tempo di fine, abbiamo che la procedura è in grado di selezionare un insieme S con n attività in tempo $\Theta(n)$.

La procedura GREEDY-ACTIVITY-SELECTOR seleziona sempre l'attività con il più piccolo tempo di fine tra i tempi di quelle attività che possono essere ancora selezionate. Pertanto, la scelta dell'attività è una scelta "greedy" nel senso che, intuitivamente, vengono lasciate il maggior numero possibile di opportunità per la selezione delle attività rimanenti. Il meccanismo di scelta greedy, quindi, è quello che massimizza la quantità di tempo che deve essere ancora assegnata.

Dimostrazione della correttezza dell'algoritmo greedy

Gli algoritmi greedy non trovano sempre la soluzione ottima di un qualsiasi problema. Tuttavia, la procedura GREEDY-ACTIVITY-SELECTOR trova sempre la soluzione ottima di qualsiasi istanza del problema della selezione di attività.

Teorema 17.1

La procedura GREEDY-ACTIVITY-SELECTOR trova sempre soluzioni di cardinalità massima nel problema della selezione di attività.

Dimostrazione. Sia $S = \{1, 2, \ldots, n\}$ l'insieme delle attività che devono essere selezionate. Dato che le attività sono ordinate per ordine crescente relativamente ai tempi di fine, abbiamo che l'attività 1 ha il più piccolo tempo di fine. Si vuole mostrare che esiste una soluzione ottima che inizia con la scelta greedy della attività numero 1.

Supponiamo che $A \subseteq S$ sia una soluzione ottima del problema della selezione di attività. Inoltre, supponiamo che le attività che appartengono all'insieme A siano ordinate in modo crescente rispetto ai loro tempi di fine e che la prima attività in A sia l'attività k. Se $k = 1$ allora l'insieme A comincia con una scelta greedy. Altrimenti, se $k \neq 1$, si dimostra che esiste una differente soluzione ottima B che inizia con una scelta greedy, ovvero con l'attività 1. Sia $B = (A - \{k\}) \cup \{1\}$. Dato che $f_1 \leq f_k$ abbiamo che gli elementi di B sono tutti disgiunti, e, quindi, B ha lo stesso numero di attività di A ed è anche una soluzione ottima del problema. Abbiamo così dimostrato che esiste sempre un assegnamento ottimo che inizia con una scelta greedy.

Inoltre, dopo avere fatto la scelta greedy dell'attività 1, il problema si riduce a trovare una soluzione ottima del problema della selezione di quelle attività in S che sono compatibili con l'attività 1. Se A è una soluzione ottima del problema originale S, allora $A' = A - \{1\}$ è una soluzione ottima del problema della selezione delle attività $S' = \{i \in S : s_i \geq f_1\}$. Qual è la ragione di questa proprietà? Se potessimo trovare una soluzione B' del problema S' con un numero di attività maggiore di quello di A', allora inserendo l'attività 1 in B' si otterrebbe una soluzione B per il problema S con un numero di attività maggiore di quello di A. Questo sarebbe una contraddizione dell'ottimalità di A. Si noti che dopo una qualunque scelta greedy si presenta sempre un problema di ottimizzazione della stessa forma del problema originale. Con un procedimento induttivo, si dimostra che la strategia che a ogni passo considera la scelta greedy permette di ottenere una soluzione ottima. ■

Esercizi

17.1-1 Si definisca un algoritmo di programmazione dinamica che risolva il problema della selezione delle attività. L'idea di base dell'algoritmo consiste nel calcolare il numero m_i di elementi dell'insieme che contiene il maggior numero di attività mutuamente compatibili tra le attività $\{1, 2, \ldots, i\}$, per $i = 1, 2, \ldots, n$. Si supponga che le attività in ingresso siano ordinate come nel caso dell'equazione (17.1). Si confronti il tempo di esecuzione della soluzione proposta con il tempo di esecuzione della procedura GREEDY-ACTIVITY-SELECTOR.

17.1-2 Supponiamo che un certo numero di sale siano condivise da un insieme di attività. Si vogliono selezionare le attività in modo tale da utilizzare il minor numero possibile di sale. Si definisca un algoritmo greedy efficiente che determini gli assegnamenti tra le attività e le sale.

(Questo problema è conosciuto anche con il nome di ***problema della colorazione di un grafo di intervalli***. Possiamo creare un grafo di intervalli nel modo seguente: i vertici del grafo rappresentano le attività, mentre gli archi mettono in relazione le attività che sono incompatibili. Determinare il minimo numero di colori necessari per colorare ogni vertice in modo tale che non esistono vertici adiacenti colorati con il medesimo colore, corrisponde a trovare il minimo numero di sale di lettura da assegnare alle attività in esame.)

17.1-3 Nel caso del problema della selezione di attività, non è vero che una qualunque strategia greedy permette di ottenere l'insieme, con il maggior numero di elementi, di attività mutamente compatibili. Si mostri un esempio in cui la strategia che seleziona l'attività con la minore durata, tra le attività che sono compatibili con le attività selezionate in precedenza, non determina la soluzione desiderata. Si consideri anche il caso della strategia che seleziona sempre l'attività che si sovrappone il meno possibile con le attività rimanenti.

17.2 Strategia greedy: concetti di base

Un algoritmo greedy permette di ottenere una soluzione ottima di un problema mediante una sequenza di decisioni. In ogni punto di scelta dell'algoritmo viene sempre presa la decisione che, al momento, appare come la migliore. Questa strategia euristica non permette di trovare sempre una soluzione ottima, ma, come abbiamo visto nel caso del problema della selezione delle attività, in molte classi di problemi è particolarmente vantaggiosa. In questo paragrafo discuteremo alcune delle proprietà peculiari dei metodi greedy.

In quale modo si può determinare se un algoritmo greedy è in grado di trovare la soluzione di un certo problema di ottimizzazione? A questa domanda non si può dare sempre una risposta precisa, tuttavia si può affermare che la maggior parte dei problemi che possono essere risolti con una strategia greedy presentano alcune caratteristiche comuni: la proprietà della scelta greedy e la sottostruttura ottima.

Proprietà della scelta greedy

La prima proprietà chiave è la ***proprietà della scelta greedy***: si può ottenere una soluzione ottima globale prendendo delle decisioni che sono ottimi locali (greedy). Questo è il punto dove gli algoritmi greedy si differenziano dagli algoritmi di programmazione dinamica. Anche nella programmazione dinamica a ogni passo vengono prese delle decisioni, ma la scelta tra le alternative può dipendere dalla soluzione dei sottoproblemi. Invece, in un algoritmo greedy si prende sempre quella decisione che al momento appare come la migliore ed in seguito si risolve il sottoproblema originato da quella scelta. La modalità di scelta di un algoritmo greedy può dipendere dalle decisioni prese fino a quel punto, ma non può dipendere da una qualunque delle scelte future o dalle soluzioni dei sottoproblemi. Pertanto, a differenza della programmazione dinamica, che risolve i sottoproblemi con una modalità bottom-up, una strategia greedy generalmente adotta una modalità top-down, dove le decisioni vengono prese una dopo l'altra con un meccanismo di scelta greedy e ogni volta si riducono le dimensioni del problema.

Naturalmente, si deve dimostrare che una strategia di scelta greedy permette di ottenere una soluzione ottima globale, e questo può richiedere una abilità ed ingegnosità non comuni. Generalmente accade che la dimostrazione consideri una soluzione ottima globale generica (questo è il caso del Teorema 17.1). Successivamente si dimostra che la soluzione può essere modificata in modo tale che il primo passo di risoluzione sia determinato da una scelta greedy. Chiaramente si deve anche dimostrare che in questo modo si ottiene sempre un problema simile al problema originale ma con dimensioni ridotte. Infine, si applica un procedimento induttivo per dimostrare che si può applicare una strategia di scelta greedy a ogni passo. Dimostrare che una scelta greedy permette di ottenere un problema simile al problema originale ma con dimensioni ridotte, significa ridurre la dimostrazione di correttezza alla dimostrazione che la soluzione ottima presenta una sottostruttura ottima.

Sottostruttura ottima

Un problema presenta una ***sottostruttura ottima*** se una soluzione ottima del problema contiene al suo interno una soluzione ottima dei sottoproblemi. Questa proprietà è di fondamentale importanza per accertare l'applicabilità sia delle tecniche di programmazione dinamica che degli algoritmi greedy. Come esempio di sottostruttura ottima si consideri nuovamente la dimostrazione del Teorema 17.1. Nella dimostrazione del teorema si mostrava che una soluzione ottima A del problema della selezione delle attività iniziava con l'attività 1, e quindi che l'insieme delle attività $A' = A - \{1\}$ era una soluzione ottima del problema della selezione delle attività per $S' = \{i \in S : s_i \geq f_1\}$.

Analisi comparata degli algoritmi greedy e della programmazione dinamica

Dato che la proprietà della sottostruttura ottima è sfruttata sia dalle strategie greedy che dalle strategie di programmazione dinamica, si potrebbe essere portati a risolvere un problema con una tecnica di programmazione dinamica anche quando è sufficiente un algoritmo greedy. Oppure, si potrebbe erroneamente pensare che una soluzione greedy funzioni perfettamente nei casi in cui è invece indispensabile una soluzione di programmazione dinamica. Per

mettere in evidenza le sottili differenze tra le due tecniche, in questo paragrafo esaminiamo due varianti di un classico problema di ottimizzazione.

Il ***problema discreto dello zaino*** (o problema dello zaino 0 - 1) è definito nel modo seguente. Un ladro durante una rapina in un negozio si trova di fronte a n articoli; l'articolo i-esimo ha un costo di v_i dollari ed un peso di w_i chilogrammi, dove sia v_i che w_i sono numeri interi. Il ladro vuole prendere gli articoli più costosi, ma il suo zaino può sopportare al massimo un peso di W chilogrammi, per un qualche intero W. Quali sono gli articoli più convenienti da rubare? (Questo problema è noto con il nome di problema dello zaino 0 - 1 perché un articolo deve essere preso o lasciato stare; il ladro non può prendere una parte di un articolo, e lo stesso articolo non può essere preso più volte).

Nel ***problema continuo dello zaino*** (o problema dello zaino generico), il ladro vuole sempre scegliere un insieme di articoli da trasportare in modo da massimizzare il valore trasportato, ma in questo caso il ladro non si trova di fronte a una scelta del tipo "prendere o lasciare", ma può scegliere di prendere parti di articoli. Un articolo nel problema dello zaino 0 - 1 può essere visto come un lingotto d'oro, mentre un articolo nel problema dello zaino generico può essere visto come della polvere d'oro.

Entrambe le formulazioni del problema dello zaino soddisfano la proprietà della sottostruttura ottima. Nel caso del problema dello zaino 0 - 1, si consideri l'elemento j di valore massimo che pesi al più W chilogrammi. Chiaramente, se si rimuove l'articolo j da questo carico si ottiene il più grande valore trasportato che pesa al massimo $W - w_j$ che il ladro può mettere assieme avendo a disposizione $n - 1$ articoli, quelli originali con l'esclusione dell'articolo j. Nel caso del problema dello zaino generico, se si toglie dal carico ottimo il peso w di una parte dell'articolo j, allora il carico rimanente è necessariamente il più grande valore trasportato che pesa al massimo $W - w$ che il ladro può mettere assieme avendo a disposizione $n - 1$ articoli e $w_j - w$ chilogrammi dell'articolo j.

Sebbene i due problemi siano alquanto simili, il problema dello zaino generico può essere risolto con una strategia greedy, mentre una strategia greedy non trova la soluzione del problema dello zaino 0 - 1. Per ottenere una soluzione del problema dello zaino generico, prima di tutto si deve calcolare il valore per unità di peso di ogni articolo: v_i/w_i. Seguendo una strategia greedy, il ladro per prima cosa prende la maggiore quantità possibile dell'articolo con il più grande valore per unità di peso. Se accade che la dotazione dell'articolo si esaurisce quando il ladro è ancora in grado di trasportare altre cose, allora il ladro prende la maggiore quantità possibile dell'articolo che, a questo punto, risulta avere il più grande valore per unità di peso. Questa operazione termina quando il ladro non è più in grado di trasportare altre cose. Pertanto, ordinando gli articoli relativamente al loro valore per unità di peso, l'algoritmo greedy viene eseguito con un tempo $O(n \lg n)$. La dimostrazione che il problema dello zaino soddisfa la proprietà della scelta greedy è lasciata come esercizio (Esercizio 17.2-1).

A questo punto vogliamo mostrare che la strategia greedy non funziona nel caso del problema dello zaino 0 - 1. Consideriamo il caso particolare di questo problema descritto in figura 17.2(a): abbiamo tre articoli e lo zaino può sopportare un peso di 50 chilogrammi. Il primo articolo ha un peso di 10 chilogrammi ed un costo di 60 dollari. Il secondo articolo ha un peso di 20 chilogrammi ed un costo di 100 dollari. Il terzo articolo ha un peso di 30 chilogrammi ed un costo di 120 dollari. Quindi, il valore per unità di peso dell'articolo 1 è di 6 dollari, che è più grande del valore per unità di peso dell'articolo 2 (5 dollari), che a sua volta è più grande del valore per unità di peso dell'articolo 3 (4 dollari). In questo caso la strategia greedy prima di tutto selezionerebbe l'articolo 1. Tuttavia, come si può facilmente vedere dall'analisi dei casi descritta nella figura 17.2(b), la soluzione ottima seleziona gli

Figura 17.2 *La strategia greedy non permette di ottenere la soluzione del problema dello zaino 0 - 1.* (a) *Il ladro deve selezionare un sottoinsieme dei tre articoli in modo tale da non superare la capacità di 50 chilogrammi dello zaino.* (b) *Il sottoinsieme ottimo contiene gli articoli 2 e 3. Tutte le soluzioni che selezionano l'articolo 1 non sono ottime, anche se l'articolo 1 ha il valore maggiore per unità di peso.* (c) *Il problema dello zaino generico ha una soluzione ottima con la strategia che seleziona gli articoli rispetto al loro costo per unità di peso.*

articoli 2 e 3, tralasciando l'articolo 1. Le due possibili soluzioni che tengono conto dell'articolo 1 sono entrambe "quasi" ottime.

Tuttavia, la strategia greedy che seleziona prima di tutto l'articolo 1 trova la soluzione ottima di questo stesso problema visto nella sua formulazione più generale, come mostrato in figura 17.2(c). La strategia che seleziona prima di tutto l'articolo 1 non trova la soluzione del problema 0 - 1 dato che il ladro non è in grado di riempire lo zaino al massimo della sua capacità, e lo spazio lasciato vuoto fa diminuire il valore effettivo (per unità di peso) del carico.

Nel caso del problema 0 - 1, prima di prendere la decisione di inserire o meno un articolo nello zaino, si deve comparare la soluzione del sottoproblema in cui lo zaino contiene quell'articolo, con la soluzione del sottoproblema in cui quell'articolo non è selezionato. Questa formulazione del problema dà origine a numerosi sottoproblemi comuni: una caratteristica distintiva della programmazione dinamica. Infatti, il problema 0 - 1 può essere risolto con tecniche di programmazione dinamica. (Cfr. Esercizio 17.2-2.)

Esercizi

17.2-1 Si dimostri che il problema dello zaino soddisfa la proprietà della scelta greedy.

17.2-2 Si descriva una soluzione di programmazione dinamica del problema dello zaino 0 - 1 tale da essere eseguita in tempo $O(nW)$, dove n è in numero degli articoli e W è la capacità di carico dello zaino del ladro.

17.2-3 Supponiamo che la sequenza degli articoli, nel problema dello zaino 0 - 1, ordinata in modo crescente rispetto al peso degli articoli, sia uguale alla sequenza degli articoli ordinata in modo decrescente rispetto al valore. Si progetti un algoritmo efficiente per trovare la soluzione ottima di questa variante del problema dello zaino. Si discuta la correttezza della soluzione proposta.

17.2-4 Il professor Mida sta percorrendo con la sua automobile il tratto della autostrada che porta da Roma a Montecarlo. La sua automobile è in grado di percorrere n chilometri con un pieno di benzina e la sua cartina riporta le distanze tra le stazioni di servizio dislocate lungo il tratto di autostrada che sta percorrendo. Il professore vorrebbe fare il minor numero possibile di soste per rifornire di benzina la sua macchina. Si progetti un algoritmo grazie al quale il professor Mida è in grado di determinare a quali stazioni di rifornimento si debba fermare; si dimostri che la strategia proposta permette di ottenere una soluzione ottima.

17.2-5 Si progetti un algoritmo efficiente che, preso un insieme di punti $\{x_1, x_2, \ldots, x_n\}$ sulla retta dei reali, determini il più piccolo insieme di intervalli chiusi di lunghezza unitaria che contenga tutti i punti in esame. Si discuta la correttezza della soluzione proposta.

* *17.2-6* Si mostri come sia possibile risolvere il problema dello zaino generico in tempo $O(n)$. Si supponga di poter considerare la soluzione del Problema 10-2.

17.3 Codici di Huffman

I codici di Huffman costituiscono una tecnica diffusa ed efficiente per la compressione di dati. Generalmente, a seconda delle caratteristiche del file che deve essere compresso, questi codici permettono di risparmiare una quantità di spazio che va dal 20% al 90%. L'algoritmo greedy, proposto da Huffman per definire la strategia ottima di rappresentazione di ogni carattere come stringa binaria, fa uso di una tabella che contiene la frequenza delle occorrenze di ciascun carattere.

Supponiamo di voler comprimere un file di dati contenente 100000 caratteri. La frequenza dell'occorrenza dei caratteri nel file è descritta dalla figura 17.3. Si noti che appaiono solamente sei caratteri distinti e che il carattere "a" appare 45000 volte.

Le strategie che permettono di rappresentare una tale quantità di informazione sono numerose. In questo paragrafo prenderemo in esame il problema della progettazione di un ***codice binario per i caratteri*** (più semplicemente ***codice***) in cui ogni carattere è rappresentato da un'unica stringa binaria. Con un ***codice a lunghezza fissa***, abbiamo bisogno di tre bit per rappresentare sei caratteri: a = 000, b = 001, ..., f = 101. Questo metodo richiede 300000 bit per codificare tutto il file. Possiamo fare di meglio?

Un ***codice a lunghezza variabile*** si comporta molto meglio di un codice a lunghezza fissa, dato che si può assegnare una parola del codice corta ai caratteri molto frequenti, e una parola lunga ai caratteri meno frequenti. La figura 17.3 mostra un codice a lunghezza variabile; la stringa binaria di un solo bit 0 rappresenta il carattere a, la stringa binaria di quattro bit 1100 rappresenta il carattere f. Questo codice richiede

	a	b	c	d	e	f
Frequenza (in migliaia)	45	13	12	16	9	5
Codice a lunghezza fissa	000	001	010	011	100	101
Codice a lunghezza variabile	0	101	100	111	1101	1100

Figura 17.3 *Il problema di codifica di caratteri. In un file di dati con 100000 caratteri sono presenti solamente i caratteri a-f, con le frequenze indicate nella figura. Se a ogni carattere è assegnata una parola di tre bit, abbiamo che il file può essere codificato con 300000 bit. Invece, con il codice a lunghezza variabile descritto nella figura, il file può essere codificato con 224000 bit.*

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224000 \text{ bit}$$

per rappresentare il file, con un risparmio approssimativo del 25%. Infatti, come vedremo nel seguito, questo è un codice ottimo per il file in esame.

Codici prefissi

In questo paragrafo prenderemo in considerazione solamente quei codici dove nessuna parola del codice è anche un prefisso di qualche altra parola del codice. Questi codici sono conosciuti con il nome di ***codici prefissi***[2]. Si può dimostrare (ma noi non lo faremo) che la compressione ottima ottenibile con un codice per i caratteri può essere sempre ottenuta con un codice prefisso. Pertanto, nonsi perde in generalità se restringiamo la nostra attenzione solamente ai codici prefissi.

I codici prefissi hanno delle buone proprietà che permettono di semplificare la codifica (compressione) e la decodifica. Generalmente, la codifica è sempre semplice per un qualunque codice di caratteri binario: si concatenano le parole che rappresentano i caratteri del file. Per esempio, con il codice di lunghezza variabile descritto nella figura 17.3, possiamo codificare il file contenente i tre caratteri abc come 0 • 101 • 100 = 0101100. Il simbolo "•" viene utilizzato per indicare la concatenazione.

L'operazione di decodifica è piuttosto semplice nel caso di un codice prefisso. Dato che nessuna parola del codice è un prefisso di una qualunque altra parola del codice, la parola con cui inizia il file codificato è identificata senza alcuna ambiguità. Pertanto, possiamo identificare la parola iniziale del codice, trasformarla nel carattere originario, rimuoverla dal file codificato e ripetere il procedimento di decodifica sulla parte rimanente del file codificato. Nel nostro esempio, la stringa 001011101 è rappresentata in modo univoco come 0•0•101•1101, che viene decodificata come aabe.

Il procedimento di decodifica richiede una rappresentazione "furba" del codice prefisso, in modo tale che la parola iniziale del codice possa essere facilmente identificata. Una rappresentazione efficiente del codice prefisso è data da alberi binari in cui le foglie rappresentano i caratteri. Il codice binario di una parola può essere interpretato come il cammino di accesso che porta dalla radice a quel carattere. In questo contesto il numero 0 significa "visita il sottoalbero sinistro", mentre 1 significa "visita il sottoalbero destro". La figura 17.4 mostra gli alberi relativi ai due codici del nostro esempio. Si noti che questi alberi non sono alberi binari di ricerca dato che le foglie non sono ordinate e, inoltre, i nodi interni non contengono chiavi.

Il codice ottimo di un file è rappresentato sempre da un albero pienamente binario dove ogni nodo, con esclusione delle foglie, ha due figli (Cfr. Esercizio 17.3-1). Nel nostro esempio, il codice a lunghezza fissa non è ottimo poiché il suo albero, mostrato nella figura 17.4(a), non è un albero pienamente binario: esistono delle parole che iniziano con 10…, ma non esiste nessuna parola che inizia con 11…. In questo paragrafo, considereremo solamente alberi pienamente binari. Pertanto, possiamo affermare che se C è l'alfabeto dei caratteri, allora l'albero del codice prefisso ottimo ha esattamente $|C|$ foglie, una per ogni carattere dell'alfabeto, e $|C| - 1$ nodi interni.

[2] Forse il termine "codici liberi da prefissi" costituirebbe un nome migliore, ma nella letteratura è utilizzato il termine "codici prefissi".

Figura 17.4 *Gli alberi che corrispondono agli schemi di codifica di figura 17.3. Ogni foglia è etichettata con un carattere e la frequenza relativa della sua occorrenza. Ogni nodo intermedio è etichettato con la somma dei pesi delle foglie dei suoi sottoalberi.* (a) *L'albero descrive il codice di lunghezza fissa* a = 000, b = 001, …, f = 101. (b) *L'albero descrive il codice prefisso ottimo* a = 0, b = 101, …, f = 1100.

A partire dall'albero T che rappresenta un codice prefisso, è abbastanza facile calcolare il numero di bit necessari per codificare il file. Per ogni carattere c nell'alfabeto C, indichiamo con $f(c)$ la frequenza (assoluta) del carattere c nel file, e con $d_T(c)$ la profondità della foglia etichettata con c. Si noti che $d_T(c)$ rappresenta anche la lunghezza della parola che codifica il carattere c. Il numero di bit necessari per codificare un file risulta

$$B(T) = \sum_{c \in C} f(c) d_T(c) \,. \tag{17.3}$$

Diciamo che questo valore descrive il **costo** dell'albero T.

Costruzione di un codice di Huffman

Huffman ha progettato un algoritmo che costruisce un codice prefisso ottimo, conosciuto con il nome di ***codice di Huffman***. L'algoritmo costruisce l'albero T che rappresenta il codice ottimo con una modalità bottom-up. Inizialmente si considera un insieme contenente $|C|$ foglie, poi, per costruire l'albero, si esegue una sequenza di $|C| - 1$ operazioni di "fusione".

Nella procedura seguente, si suppone che C sia un alfabeto di n caratteri, e che ogni carattere $c \in C$ sia un oggetto con una frequenza definita da $f[c]$. Una coda con priorità, con chiavi basate su f, viene utilizzata per identificare i due oggetti con la frequenza più piccola da fondere insieme. Il risultato dell'operazione di fusione è un nuovo oggetto la cui frequenza è data dalla somma delle frequenze degli oggetti che sono stati fusi insieme.

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
    do    z ← ALLOCATE-NODE()
          x ← left[z] ← EXTRACT-MIN(Q)
          y ← right[z] ← EXTRACT-MIN(Q)
```

```
7        f[z] ← f[x] + f[y]
8        INSERT(Q, z)
9    return EXTRACT-MIN(Q)
```

Per il nostro esempio, l'algoritmo di Huffman procede come descritto nella figura 17.5. Dato che l'alfabeto contiene solamente sei caratteri, la dimensione iniziale della coda Q è $n = 6$, inoltre sono necessari cinque passi di fusione per ultimare la costruzione dell'albero. L'albero così costruito rappresenta il codice prefisso ottimo. La parola associata a ogni carattere è definita dalla sequenze delle etichette degli archi che portano dalla radice alla foglia etichettata con quel carattere.

La linea 2 assegna alla coda con priorità Q i caratteri in C. Il ciclo **for** (linee 3-8) rimuove dalla coda i due nodi x e y con la frequenza più bassa, ed inserisce nella coda un nuovo nodo z che è il risultato della fusione di x e y (linea 7). Il nodo z ha x come figlio sinistro e y come figlio destro. (Questo ordine è arbitrario: scambiando tra loro i figli di sinistra e di destra di ogni nodo si ottiene un codice differente ma con lo stesso costo.) Dopo $n - 1$ operazioni di

Figura 17.5 *I passi dell'algoritmo di Huffman per le frequenze descritte nella figura 17.3. Ogni parte della figura mostra il contenuto della coda ordinata in modo crescente rispetto alla frequenza. A ogni passo dell'algoritmo, i due alberi con la frequenza minore vengono fusi assieme. Le foglie sono rappresentate da rettangoli etichettati con il carattere e la sua frequenza. I nodi interni sono rappresentati da cerchi etichettati con la somma delle frequenze dei figli. Un arco che mette in relazione un nodo interno con uno dei suoi figli è etichettato con 0 se è l'arco del figlio di sinistra, 1 se è l'arco del figlio di destra. La parola di codice per un carattere è data dalla sequenza delle etichette dei nodi che portano dalla radice alla foglia etichettata con quel carattere.* **(a)** *L'insieme dei 6 nodi iniziali, uno per ogni carattere.* **(b)-(e)** *I passi intermedi.* **(f)** *L'albero finale.*

fusione, la procedura restituisce come risultato (linea 9) l'unico nodo che rimane nella coda: la radice dell'albero del codice.

L'analisi del tempo di esecuzione dell'algoritmo di Huffman assume che la coda Q sia realizzata con un heap binario (Capitolo 7). Per un insieme C di n caratteri, la fase di inizializzazione della coda Q (linea 2) può essere eseguita, con la procedura BUILD-HEAP descritta nel paragrafo 7.3, in un tempo $O(n)$. Il ciclo **for** (linee 3-8) viene eseguito per un totale di $n - 1$ volte e, dato che ogni operazione dello heap ha tempo di esecuzione $O(\lg n)$, il contributo del ciclo al tempo di esecuzione dell'algoritmo risulta $O(n \lg n)$. In conclusione, il tempo di esecuzione della procedura HUFFMAN su un insieme di n caratteri è $O(n \lg n)$.

Correttezza dell'algoritmo di Huffman

Per dimostrare che l'algoritmo greedy di Huffman è corretto, mostriamo che il problema della costruzione del codice prefisso ottimo soddisfa le proprietà della scelta greedy e della sottostruttura ottima. Il lemma che segue dimostra che la proprietà della scelta greedy è soddisfatta.

Lemma 17.2

Sia C un alfabeto dove ogni carattere c ha una frequenza $f[c]$. Siano x e y due caratteri di C con la frequenza minore. Allora esiste sempre un codice prefisso ottimo per C dove le parole associate a x e y hanno la stessa lunghezza e differiscono solamente per l'ultimo bit.

Dimostrazione. La dimostrazione considera l'albero T che rappresenta il codice prefisso ottimo e lo trasforma in un altro albero, che a sua volta rappresenta un diverso codice prefisso ottimo, in modo tale che nel nuovo albero i caratteri x e y siano associati a foglie che hanno lo stesso padre e profondità massima. Se questa trasformazione è possibile allora significa che le parole del codice per x e y avranno la stessa lunghezza e saranno diverse solo per l'ultimo bit.

Siano b e c due foglie dell'albero T figlie dello stesso padre e di profondità massima. Senza perdere in generalità possiamo supporre $f[b] \le f[c]$ e $f[x] \le f[y]$. Per le ipotesi fatte $f[x]$ e $f[y]$ sono le foglie con le frequenze minori, $f[b]$ e $f[c]$ sono due frequenze arbitrarie e pertanto $f[x] \le f[b]$ e $f[y] \le f[c]$.

La trasformazione cercata è mostrata nella figura 17.6. Per prima cosa si scambiano in T le posizioni di b e x, ottenendo in questo modo un nuovo albero T', poi si scambiano in T' le

Figura 17.6 *Una descrizione del passo cruciale nella dimostrazione del lemma 17.2. Le foglie b e c dell'albero ottimo T sono due foglie con il medesimo padre a profondità massima. Le foglie x e y sono le prime due foglie che l'algoritmo di Huffman fonde assieme. Queste due foglie appaiono in una posizione qualunque in T. Le foglie b e x sono scambiate ottenendo in questo modo l'albero T'. Poi vengono scambiate le foglie c e y ottenendo l'albero T''. Dato che gli scambi non fanno aumentare il costo, l'albero T'' è ancora un albero ottimo.*

posizioni di c e y, ottenendo un nuovo albero T''. Grazie all'equazione (17.3) abbiamo che la differenza del costo di T e T' risulta:

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f[x]d_T(x) + f[b]d_T(b) - f[x]d_{T'}(x) - f[b]d_{T'}(b) \\ &= f[x]d_T(x) + f[b]d_T(b) - f[x]d_T(b) - f[b]d_T(x) \\ &= (f[b] - f[x])(d_T(b) - d_T(x)) \\ &\geq 0 , \end{aligned}$$

dato che sia $f[b] - f[x]$ sia $d_T(b) - d_T(x)$ sono entrambi maggiori o uguali a zero. Più precisamente, il risultato di $f[b] - f[x]$ non è un numero negativo perché x è la foglia con la minore frequenza. Analogamente, il risultato di $d_T(b) - d_T(x)$ non è un numero negativo perché b è una foglia di profondità massima in T. Con un ragionamento simile abbiamo che lo scambio tra y e c non fa aumentare il costo, pertanto la differenza $B(T') - B(T'')$ è maggiore o uguale a zero. In conclusione abbiamo che $B(T'') \leq B(T)$, e dato che T è ottimo si ha che $B(T) \leq B(T'')$, e il tutto implica che $B(T'') = B(T)$. Quindi, T'' è un albero ottimo in cui x e y sono foglie dello stesso padre di profondità massima. Questo conclude la dimostrazione del lemma. ■

Il lemma 17.2 dimostra che il procedimento, che costruisce un albero ottimo mediante una sequenza di operazioni di fusione, può sempre incominciare con una scelta greedy dei due caratteri con la frequenza più piccola da fondere assieme. A questo punto ci si domanda: per quale motivo questo tipo di scelta costituisce una scelta greedy? Il costo di una operazione di fusione può essere definito come la somma delle frequenze dei due oggetti che sono fusi assieme. L'Esercizio 17.3-3 dimostra che il costo totale della costruzione dell'albero è dato dalla somma dei costi delle operazioni di fusione che sono necessarie per costruirlo. A ogni passo la procedura Huffman, tra tutte le possibili operazioni di fusione, sceglie sempre quella operazione che ha il costo minore.

Il lemma seguente dimostra che il problema della costruzione del codice prefisso ottimo soddisfa la proprietà della sottostruttura ottima.

Lemma 17.3

Sia T un albero binario pieno che rappresenta un codice prefisso ottimo per un alfabeto C con funzione di frequenza $f[c]$, per ogni carattere $c \in C$. Siano x e y due caratteri qualsiasi che appaiono come foglie dello stesso padre z in T. Se si considera z come un carattere la cui frequenza è $f[z] = f[x] + f[y]$, allora l'albero $T' = T - \{x, y\}$ rappresenta un codice prefisso ottimo per l'alfabeto $C' = C - \{x, y\} \cup \{z\}$.

Dimostrazione. Prima di tutto mostriamo che il costo $B(T)$ dell'albero T può essere espresso in termini del costo $B(T')$ dell'albero T', esaminando le componenti dei costi nell'equazione (17.3). Per ogni $c \in C - \{x, y\}$, abbiamo che $d_T(c) = d_{T'}(c)$, e di conseguenza $f[c]d_T(c) = f[c]d_{T'}(c)$. Dato che $d_T(x) = d_T(y) = d_{T'}(z) + 1$, abbiamo che

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])(d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + (f[x] + f[y]) , \end{aligned}$$

da cui possiamo concludere che

$$B(T) = B(T') + f[x] + f[y] .$$

Se T' rappresenta un codice prefisso non ottimo per l'alfabeto C', allora esiste un albero T'' le cui foglie sono i caratteri di C', e tale che $B(T'') < B(T')$. Poiché z è assimilato a un carattere di C', allora è una foglia di T''. Se aggiungiamo a T'' come figli di z sia x che y, otteniamo un codice prefisso per C con un costo $B(T'') + f[x] + f[y] < B(T)$. Questo è in contraddizione con l'ipotesi che T sia ottimo per C. Pertanto, T' deve essere ottimo per l'alfabeto C'. ■

Teorema 17.4

La procedura HUFFMAN restituisce un codice prefisso ottimo.

Dimostrazione. Conseguenza immediata dei lemmi 17.2 e 17.3. ■

Esercizi

17.3-1 Si dimostri che un albero binario non pieno non può rappresentare un codice prefisso ottimo.

17.3-2 Qual è il codice di Huffman ottimo per il seguente insieme di frequenze basato sui primi otto numeri di Fibonacci?

a : 1 b : 1 c : 2 d : 3 e : 5 f : 8 g : 13 h : 21

Si può generalizzare la soluzione per trovare il codice ottimo nel caso di frequenze basate sui primi n numeri di Fibonacci?

17.3-3 Si dimostri che il costo di un albero che rappresenta un codice può essere anche calcolato come la somma delle frequenze associate ai nodi interni.

17.3-4 Si supponga di considerare un insieme di caratteri ordinati in modo tale che le loro frequenze siano non crescenti. Si dimostri che un codice ottimo deve avere le lunghezze delle parole di codice dei caratteri non decrescenti.

17.3-5 Sia $C = \{0, 1, \ldots, n-1\}$ un insieme di caratteri. Si dimostri che un qualunque codice prefisso ottimo per C può essere rappresentato da una sequenza di bit di lunghezza $2n - 1 + n\lceil \lg n \rceil$.

(*Suggerimento*: conviene usare $2n - 1$ bit per specificare la struttura dell'albero come se fosse generata da una visita dell'albero.)

17.3-6 Si generalizzi l'algoritmo di Huffman a parole ternarie (cioè parole costruite con i simboli 0, 1 e 2). Si dimostri che questa generalizzazione restituisce codici ternari ottimi.

17.3-7 Supponiamo che un file di dati contenga una sequenza di caratteri di 8 bit in modo tale che tutti i 256 caratteri abbiano frequenze poco diverse: supponiamo che la

frequenza massima sia minore del doppio della frequenza minima. Si dimostri che, in questo caso, la codifica di Huffman non è più efficiente della codifica che utilizza un normale codice di lunghezza fissa di 8 bit.

* *17.3-8* Si dimostri che nessuno schema di compressione di dati è in grado di comprimere un file neppure di un singolo bit, se i caratteri (di 8 bit) sono scelti a caso. (*Suggerimento*: si confronti il numero dei file con il numero dei possibili file codificati.)

* 17.4 Fondamenti teorici dei metodi greedy

In questo paragrafo descriveremo brevemente l'affascinante teoria che sta alla base degli algoritmi greedy. Questa teoria si dimostra particolarmente utile nei casi in cui si vuole dimostrare che un algoritmo greedy trova la soluzione ottima. La teoria che andiamo a esporre coinvolge delle strutture combinatorie conosciute con il nome di "matroidi". Molti problemi di interesse pratico possono essere analizzati con l'ausilio di questa teoria, anche se la teoria non è in grado di coprire tutte le possibili classi di problemi risolubili con tecniche greedy (per esempio non si può applicare al problema della selezione delle attività del paragrafo 17.1 o al problema dei codici di Huffman del paragrafo 17.3). Inoltre, questa teoria si è sviluppata assai rapidamente ed è stata estesa a molte altre applicazioni, come accennato nelle note alla fine del capitolo.

17.4.1 Matroidi

Un ***matroide*** è una coppia ordinata $M = (S, \mathcal{I})$ che soddisfa le seguenti condizioni.

1. S è un insieme finito non vuoto.
2. $\mathcal{I}$ è una famiglia non vuota di sottoinsiemi di S, i sottoinsiemi di S ***indipendenti***, tale che se $B \in \mathcal{I}$ e $A \subseteq B$ allora anche $A \in \mathcal{I}$. Diciamo che $\mathcal{I}$ è ***ereditario*** se soddisfa la precedente proprietà. Si noti che l'insieme vuoto $\varnothing$ appartiene necessariamente alla famiglia $\mathcal{I}$.
3. Se $A \in \mathcal{I}$, $B \in \mathcal{I}$ e $|A| < |B|$, allora esiste un qualche elemento $x \in B - A$ tale che $A \cup \{x\} \in \mathcal{I}$. In questo caso diciamo che M soddisfa la ***proprietà di scambio***.

La parola "matroide" è dovuta a Hassler Whitney che ha studiato i ***matroidi matriciali***, dove gli elementi di S sono le colonne di una certa matrice e gli elementi di $\mathcal{I}$ sono gli insiemi di colonne linearmente indipendenti (nel senso usuale della teoria delle matrici). È facile dimostrare che questo tipo di strutture definiscono un matroide (Cfr. Esercizio 17.4-2).

Per avere un altro esempio di matroide, si consideri il ***matroide grafico*** $M_G = (S_G, \mathcal{I}_G)$ definito in termini di un grafo non orientato $G = (V, E)$ nel modo seguente:

- L'insieme S_G è l'insieme E degli archi di G.
- Se A è un sottoinsieme di E, allora $A \in \mathcal{I}_G$ se e solo se A non contiene cicli. In altri termini, un insieme di archi è indipendente se e solo se forma una foresta.

Il matroide grafico M_G è strettamente collegato al problema del minimo albero di copertura, che verrà analizzato in dettaglio nel Capitolo 24.

Teorema 17.5

Se G è un grafo non orientato, allora $M_G = (S_G, \mathcal{I}_G)$ è un matroide.

Dimostrazione. Chiaramente, $S_G = E$ è un insieme finito. Inoltre, $\mathcal{I}_G$ è ereditario dato che un sottoinsieme di una foresta è ancora una foresta. In altri termini, non si possono creare dei cicli se si rimuove un arco da un insieme di archi che non contengono cicli.

A questo punto si deve solamente dimostrare che M_G soddisfa la proprietà di scambio. Supponiamo che A e B siano due foreste di G e che $|B| > |A|$. Quindi, A e B sono due insiemi di archi che non contengono cicli e tali che l'insieme B contiene un maggior numero di archi dell'insieme A.

Grazie al Teorema 5.2 sappiamo che una foresta con k archi e $|V|$ vertici contiene esattamente $|V| - k$ alberi. (Un modo alternativo di dimostrare questa proprietà è il seguente. Inizialmente si considerino $|V|$ alberi senza archi. Poi, ogni arco che viene aggiunto alla foresta fa diminuire di una unità il numero degli alberi.) Pertanto, la foresta A contiene $|V| - |A|$ alberi, e la foresta B contiene $|V| - |B|$ alberi.

Dato che la foresta B ha un numero minore di alberi della foresta A, allora la foresta B deve contenere un qualche albero T i cui vertici appartengono a due alberi distinti della foresta A. Inoltre, dato che T è connesso, allora deve necessariamente contenere un arco (u, v) tale che i vertici u e v appartengono a due alberi distinti nella foresta A. L'arco (u, v) collega due vertici di due alberi distinti nella foresta A e quindi può essere aggiunto alla foresta A senza creare un ciclo. Pertanto, M_G soddisfa la proprietà di scambio. Questo conclude la dimostrazione che M_G è un matroide. ■

Sia $M = (S, \mathcal{I})$ un matroide, diciamo che un elemento $x \notin A$ è una ***estensione*** di A in $\mathcal{I}$ se x può essere aggiunto a A ottenendo in questo modo un insieme che soddisfa la proprietà di indipendenza. Quindi, x è una ***estensione*** di A se $A \cup \{x\} \in \mathcal{I}$. Per esempio, consideriamo un matroide grafico M_G. Se A è un insieme di archi indipendente, allora l'arco e è una estensione di A se e solo se e non appartiene a A, e l'aggiunta dell'arco e ad A non crea dei cicli.

Sia A un insieme indipendente di un matroide M; diciamo che A è ***massimale*** se non ammette estensioni. Quindi, un insieme A è massimale se non è contenuto in nessun altro insieme indipendente. La seguente proprietà è utile in molte circostanze.

Teorema 17.6

Tutti i sottoinsiemi indipendenti massimali di un matroide M hanno la stessa cardinalità.

Dimostrazione. Procediamo per assurdo. Supponiamo che A sia un sottoinsieme di M indipendente e massimale, e che esista un altro sottoinsieme B di M, indipendente e massimale, più grande di A. Questo significa che, per la proprietà di scambio, A è estendibile per un qualche elemento $x \in B - A$, ottenendo in questo modo un sottoinsieme indipendente $A \cup \{x\}$ più grande di A. Chiaramente questo è un assurdo dato che per ipotesi A è massimale. ■

Per avere una idea di questo teorema, consideriamo il matroide grafico M_G nel caso di un grafo non orientato e connesso G. Ogni sottoinsieme di M_G indipendente e massimale deve essere un albero libero, con esattamente $|V| - 1$ lati e tale da collegare tutti i vertici di G. Un albero con queste caratteristiche è chiamato ***albero di copertura*** di G.

Diciamo che un matroide $M = (S, \mathcal{I})$ è ***pesato*** se esiste una funzione peso w che assegna un valore, strettamente maggiore di zero, a ogni elemento $x \in S$. La funzione peso w può essere estesa ai sottoinsiemi di S mediante la sommatoria:

$$w(A) = \sum_{x \in A} w(x)$$

per un qualunque $A \subseteq S$. Per esempio, se si considera come funzione peso di un matroide grafico la funzione che associa a ogni arco la sua lunghezza $w(e)$, allora $w(A)$ restituisce la lunghezza totale degli archi che appartengono all'insieme di archi A.

17.4.2 Algoritmi greedy su un matroide pesato

Un gran numero di problemi, che possono essere risolti con un metodo greedy, possono essere formulati in termini del problema di trovare il sottoinsieme indipendente di massimo peso in un matroide pesato. Più precisamente, quest'ultimo problema è così formulato: sia $M = (S, \mathcal{I})$ un matroide pesato, si vuole trovare un insieme indipendente $A \in \mathcal{I}$ in modo tale che $w(A)$ sia massimo. Un insieme indipendente di peso massimo è detto un sottoinsieme ***ottimo*** del matroide. Dato che il peso $w(x)$ di ogni elemento $x \in S$ è maggiore di zero, un sottoinsieme ottimo è sempre un sottoinsieme indipendente e massimale: nel risolvere questo problema è necessario (ma non sufficiente) scegliere A più grande possibile.

Per esempio, nel ***problema del minimo albero di copertura***, si considera un grafo connesso non orientato $G = (V, E)$ ed una funzione lunghezza w, che assegna una lunghezza $w(e)$ maggiore di zero a ogni arco e. In questo problema si chiede di trovare un sottoinsieme degli archi del grafo in modo che tutti i vertici del grafo siano connessi e che abbia la più piccola lunghezza totale. Per vedere questo problema come il problema che trova un sottoinsieme ottimo di un matroide, basta considerare il matroide pesato M_G, dove la funzione peso w' è definita nel modo seguente: $w'(e) = w_0 - w(e)$, e w_0 è maggiore della più grande lunghezza degli archi. Nel matroide pesato così ottenuto, tutti i pesi sono maggiori di zero ed un sottoinsieme ottimo corrisponde all'albero di copertura con la minima lunghezza del grafo originale. Più precisamente, ogni sottoinsieme A, indipendente e massimale, corrisponde a un albero di copertura e, dato che per ogni sottoinsieme A indipendente e massimale vale che

$$w'(A) = (|V| - 1)w_0 - w(A)$$

allora il sottoinsieme indipendente che massimizza $w'(A)$ deve necessariamente minimizzare $w(A)$. Pertanto, un qualunque algoritmo che sia in grado di trovare un sottoinsieme ottimo A in un matroide arbitrario è anche in grado di risolvere il problema del minimo albero di copertura.

Nel Capitolo 24 vengono presentati un certo numero di algoritmi che risolvono il problema del minimo albero di copertura; in questo paragrafo presentiamo un algoritmo greedy che funziona correttamente per un generico matroide pesato. L'algoritmo prende in ingresso un matroide pesato $M = (S, \mathcal{I})$, con la relativa funzione peso w e restituisce un sottoinsieme

ottimo A. Nella nostra procedura le componenti di M vengono indicate con $S[M]$ e $\mathcal{I}[M]$, mentre w indica la funzione peso. L'algoritmo è un algoritmo greedy dato che prende in esame, singolarmente ed in ordine non crescente rispetto al peso, ogni elemento $x \in S$, e successivamente aggiunge l'elemento in esame all'insieme A nel caso in cui $A \cup \{x\}$ risulti indipendente.

```
GREEDY(M, w)
1  A ← ∅
2  ordinare S[M] in ordine non crescente rispetto al peso w
3  for ciascun x ∈ S[M] nell'ordine determinato al passo 2
4      do if A ∪ {x} ∈ I[M]
5            then A ← A ∪ {x}
6  return A
```

Gli elementi di S vengono considerati uno dopo l'altro in ordine non crescente rispetto al loro peso. L'elemento corrente x viene inserito in A solo se il suo inserimento non fa perdere la proprietà di indipendenza. Nel caso contrario x viene scartato. Dato che, per la definizione di matroide, l'insieme vuoto è indipendente, e dato che x viene aggiunto a A solo nel caso in cui $A \cup \{x\}$ risulti indipendente, allora con un ragionamento induttivo si dimostra che il sottoinsieme A è sempre un sottoinsieme indipendente. Pertanto, la procedura GREEDY restituisce sempre un sottoinsieme indipendente. Tra poco dimostreremo che A è un sottoinsieme con peso massimo e di conseguenza è un sottoinsieme ottimo.

Il tempo di esecuzione della procedura GREEDY si ottiene facilmente. Sia $n = |S|$. La fase di ordinamento della procedura richiede tempo $O(n \lg n)$. La linea 4 è eseguita esattamente n volte, una volta per ogni elemento di S. A ogni esecuzione della linea 4 si deve controllare se l'insieme $A \cup \{x\}$ è indipendente. Assumendo che un singolo controllo richieda tempo $O(f(n))$, allora la procedura GREEDY viene eseguita in tempo $O(n \lg n + nf(n))$.

A questo punto dimostriamo che la procedura GREEDY restituisce un sottoinsieme ottimo.

Lemma 17.7 (I matroidi soddisfano la proprietà della scelta greedy)

Sia $M = (S, \mathcal{I})$ un matroide pesato con funzione peso w, tale che S sia ordinato in modo non crescente rispetto al peso degli elementi. Sia x il primo elemento di S per cui $\{x\}$ sia indipendente, nel caso in cui un tale x esista. Se esiste x allora esiste anche un sottoinsieme ottimo A che contiene x.

Dimostrazione. Se l'elemento x richiesto non esiste, allora l'unico sottoinsieme indipendente è l'insieme vuoto e quindi abbiamo concluso la dimostrazione. Nel caso contrario, sia B un qualunque sottoinsieme ottimo non vuoto. Assumiamo che $x \notin B$, poiché altrimenti possiamo concludere la dimostrazione con $A = B$.

Affermiamo che nessun elemento di B ha un peso maggiore di $w(x)$. Per dimostrare questo asserto si noti che $y \in B$ implica che $\{y\}$ è indipendente dato che $B \in \mathcal{I}$ ed $\mathcal{I}$ è ereditario. La scelta di x assicura che $w(x) \geq w(y)$, per ogni $y \in B$.

Allora costruiamo l'insieme A nel modo seguente. Inizialmente poniamo $A = \{x\}$. Per le ipotesi fatte su x, abbiamo che A è indipendente. Successivamente, grazie alla proprietà di scambio, selezioniamo un elemento di B che può essere aggiunto ad A senza perdere la proprietà di indipendenza. Ripetiamo questa operazione finché $|A| = |B|$. Alla fine di questa sequenza di operazioni abbiamo che $A = (B - \{y\}) \cup \{x\}$, per un qualche $y \in B$. Quindi

$$\begin{aligned} w(A) &= w(B) - w(y) + w(x) \\ &\geq w(B) . \end{aligned}$$

Poiché B è ottimo allora anche A deve essere ottimo e, dato che $x \in A$, il lemma è dimostrato. ■

A questo punto dimostriamo che se un elemento non è una possibile scelta iniziale, allora quell'elemento nel seguito non potrà mai diventare una scelta possibile.

Lemma 17.8

Sia $M = (S, \mathcal{I})$ un matroide. Sia x un elemento di S tale che x non è l'estensione dell'insieme vuoto $\varnothing$. Allora x non è l'estensione di un qualunque sottoinsieme indipendente A di S.

Dimostrazione. Procediamo per assurdo. Supponiamo che x sia una estensione di A ma non di $\varnothing$. Dato che x è una estensione di A, abbiamo che $A \cup \{x\}$ è indipendente. Dato che $\mathcal{I}$ è ereditario, anche $\{x\}$ deve essere indipendente. Ma questo contraddice l'ipotesi che x non sia una estensione dell'insieme vuoto $\varnothing$. ■

Il lemma 17.8 dimostra che un qualunque elemento che non può essere selezionato immediatamente, non potrà mai essere utilizzato. Pertanto, la procedura GREEDY non commette mai un errore se non considera gli elementi di S che non sono estensioni di $\varnothing$.

Lemma 17.9 (I matroidi soddisfano la proprietà della sottostruttura ottima)

Sia x il primo elemento selezionato dalla procedura GREEDY con il matroide pesato $M = (S, \mathcal{I})$. Il problema di trovare un sottoinsieme indipendente di massimo peso che contiene x è il problema di trovare un sottoinsieme indipendente di massimo peso nel matroide pesato $M' = (S', \mathcal{I}')$ dove

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in \mathcal{I}\} , \\ \mathcal{I}' &= \{B \subseteq S - \{x\} : B \cup \{x\} \in \mathcal{I}\} \end{aligned}$$

e la funzione peso di M' è la funzione peso di M, ristretta all'insieme S'. (Diciamo che M' è la *contrazione* di M attraverso x.)

Dimostrazione. Sia A un qualunque sottoinsieme indipendente di massimo peso del matroide M contenente x. Allora, $A' = A - \{x\}$ è un sottoinsieme indipendente di M'. Analogamente, un qualunque sottoinsieme indipendente A' del matroide M' definisce un sottoinsieme indipendente $A = A' \cup \{x\}$ del matroide M. In entrambi i casi vale che $w(A) = w(A') + w(x)$, e quindi una soluzione di massimo peso in M che contiene x determina una soluzione di massimo peso in M' e viceversa. ■

Teorema 17.10 (Correttezza dell'algoritmo greedy sui matroidi)

Sia $M = (S, \mathcal{I})$ un matroide pesato con funzione peso w. Allora la chiamata GREEDY(M, w) restituisce un sottoinsieme ottimo.

Dimostrazione. Per il lemma 17.8 tutti gli elementi che non sono estensione di $\varnothing$ possono essere trascurati: non saranno mai utili per costruire la soluzione ottima. Dopo aver selezionato il primo elemento x, il lemma 17.7 garantisce che la procedura GREEDY non compirà mai un errore a seguito dell'operazione che inserisce x nell'insieme A; infatti esiste sempre un sottoinsieme ottimo che contiene x. Infine, il lemma 17.9 garantisce che il problema che rimane da risolvere è il problema di trovare un sottoinsieme ottimo nel matroide M' che è la contrazione di M attraverso x. Tutte le operazioni eseguite dalla procedura GREEDY, dopo l'assegnamento di $\{x\}$ ad A, possono essere viste come operazioni sul matroide M', dato che per ogni $B \in \mathcal{I}'$ vale che B è indipendente in M' se e solo se $B \cup \{x\}$ è indipendente in M. Come risultato di queste operazioni, la procedura Greedy restituirà un sottoinsieme indipendente di massimo peso in M'. Pertanto, il risultato globale della procedura sarà un sottoinsieme indipendente di massimo peso in M. ■

Esercizi

17.4-1 Si dimostri che la coppia $(S, \mathcal{I}_k)$ è un matroide, dove S è un qualunque insieme finito e $\mathcal{I}_k$ è l'insieme di tutti i sottoinsiemi di S con cardinalità al più k, con $k \leq |S|$.

* *17.4-2* Sia T una matrice $n \times n$ a valori reali. Si dimostri che la coppia $(S, \mathcal{I})$, dove S è l'insieme delle colonne di T e $A \in \mathcal{I}$, se e solo se le colonne in A sono linearmente indipendenti, è un matroide.

* *17.4-3* Si dimostri che se $(S, \mathcal{I})$ è un matroide, allora $(S, \mathcal{I}')$, dove $\mathcal{I}' = \{A' : S - A'$ contiene almeno un insieme massimale $A \in \mathcal{I}\}$, è un matroide. In altri termini, gli insiemi indipendenti e massimali di $(S, \mathcal{I}')$ sono esattamente il complemento insiemistico degli insiemi indipendenti e massimali di $(S, \mathcal{I})$.

* *17.4-4* Sia S un insieme finito e sia $S_1, S_2, \ldots, S_k$ una partizione di S. Definiamo la struttura $(S, \mathcal{I})$ imponendo la condizione che $\mathcal{I} = \{A : |A \cap S_i| \leq 1$ per $i = 1, 2, \ldots, k\}$. Si dimostri che $(S, \mathcal{I})$ è un matroide. In altri termini, l'insieme degli insiemi A che contengono al più un elemento in ogni blocco della partizione costituisce l'insieme degli insiemi indipendenti del matroide.

17.4-5 Si mostri come trasformare la funzione peso di un problema relativo a un matroide pesato, se il problema richiede di trovare un sottoinsieme indipendente massimale di *minimo peso*, in modo da ottenere il problema classico del matroide pesato. Si discuta la correttezza della trasformazione.

* 17.5 Un problema di scheduling

Un problema interessante che può essere risolto con l'uso dei matroidi è il problema della definizione dell'ordine di esecuzione ottimo di programmi di durata unitaria. In questo problema si hanno dei programmi (ogni programma richiede una unità di tempo di esecuzione) che devono essere eseguiti su un unico processore. Ogni programma deve essere eseguito entro una certa scadenza e se la scadenza non è rispettata scatta una penalità. A una analisi superficiale questo problema appare particolarmente complicato, tuttavia un semplice algoritmo greedy è in grado di trovarne la soluzione.

Un ***programma di durata unitaria*** è un programma che richiede esattamente una unità di tempo per essere eseguito completamente. Dato un insieme finito S di programmi di durata unitaria, una ***sequenza di esecuzione*** (in inglese ***schedule***) per S è una permutazione di S che specifica l'ordine di esecuzione dei programmi. Il primo programma nella sequenza di esecuzione è attivato al tempo 0 e termina la sua esecuzione al tempo 1, il secondo programma è attivato al tempo 1 e termina al tempo 2, e così via.

Il ***problema dell'ordinamento su un unico processore di programmi di durata unitaria con scadenze e penalità*** è così definito:

- un insieme $S = \{1, 2, \ldots, n\}$ di n programmi di durata unitaria;
- un insieme di n ***scadenze*** $d_1, d_2, \ldots, d_n$, tale che $1 \le d_i \le n$, per ogni d_i;
- un insieme di n pesi o ***penalità*** $w_1, w_2, \ldots, w_n$, dove ogni penalità è un numero maggiore o uguale a zero. Il programma i incorre nella penalità w_i se termina dopo la sua scadenza d_i e invece non scatta alcuna penalità se il programma i termina entro la sua scadenza.

Si vuole trovare una sequenza di esecuzione (o ordinamento) dei programmi di S che minimizza la somma delle penalità dei programmi che non rispettano la scadenza.

Consideriamo una qualunque sequenza di esecuzione. Diciamo che un programma è in ***ritardo*** nella sequenza se termina dopo la sua scadenza, altrimenti diciamo che il programma è in ***anticipo***. Una generica sequenza di esecuzione può essere sempre trasformata in una forma detta ***forma early-first*** dove i programmi in anticipo precedono i programmi in ritardo. È evidente che una qualunque sequenza di esecuzione può essere portata in una forma early-first, infatti se un programma in ritardo x precede un programma in anticipo y possiamo scambiare le posizioni di x e y nella sequenza. Nella sequenza così ottenuta x è ancora in ritardo e y è ancora in anticipo.

In modo analogo possiamo affermare che una sequenza di esecuzione qualsiasi può essere sempre trasformata in ***forma canonica***: nella forma canonica i programmi in anticipo precedono i programmi in ritardo, ed i programmi in anticipo sono ordinati per scadenze non decrescenti. Per dimostrare la validità della affermazione precedente prima di tutto si trasforma la sequenza in forma early-first. Successivamente, si scambiano nella sequenza le posizioni di due programmi in anticipo i e j, che terminano rispettivamente ai tempi k e $k + 1$, e tali che $d_j < d_i$. Prima dello scambio il programma j è in anticipo, quindi $k + 1 \le d_j$. Pertanto, per le ipotesi fatte abbiamo che $k + 1 < d_i$ e, quindi, dopo lo scambio il programma i è ancora in anticipo. Chiaramente, nella sequenza così ottenuta il programma j è ancora in anticipo. Questo procedimento è iterato fino a ottenere una sequenza in cui i programmi in anticipo sono ordinati per scadenze non decrescenti.

Come diretta conseguenza abbiamo che la ricerca di un'ordinamento ottimo si riduce a trovare un insieme A di programmi in anticipo nell'ordinamento ottimo. Dopo aver determinato l'insieme A, la sequenza di esecuzione si ottiene ordinando per scadenze non decrescenti i programmi che appartengono all'insieme A. Successivamente, si considerano i programmi in ritardo $(S - A)$ in un ordine qualunque; in questo modo si ottiene un ordine canonico della sequenza di esecuzione ottima.

Diciamo che un insieme di programmi è ***indipendente*** se esiste una sequenza di esecuzione per questi programmi dove nessun programma è in ritardo. È chiaro che l'insieme dei programmi in anticipo di un certa sequenza di esecuzione definisce un insieme di programmi indipendenti. Con $\mathcal{I}$ indichiamo l'insieme di tutti gli insiemi di programmi indipendenti.

A questo punto prendiamo in esame il problema di stabilire se un insieme A di programmi è indipendente. Indichiamo con $N_t(A)$ $(t = 1, 2, \ldots, n)$ il numero di programmi in A la cui scadenza è minore o uguale a t.

Lemma 17.11

Per ogni insieme A di programmi le seguenti proprietà sono equivalenti.

1. L'insieme A è indipendente.
2. Per $t = 1, 2, \ldots, n$, abbiamo che $N_t(A) \le t$.
3. Se i programmi che appartengono a A sono ordinati per scadenze crescenti, allora nessun programma è in ritardo.

Dimostrazione. Chiaramente, se $N_t(A) > t$, per un qualche t, allora non abbiamo modo alcuno di definire una sequenza di esecuzione dove nessun programma di A è in ritardo. Infatti, prima del tempo t devono finire più di t programmi. Come conseguenza abbiamo che (1) implica (2). A questo punto supponiamo che (2) sia verificata. Dimostriamo che anche (3) è verificata. Infatti non è possibile "bloccarsi" quando i programmi sono ordinati per scadenze crescenti: la proprietà (2) implica che la i-esima scadenza più grande è al massimo i. Infine, è banalmente vero che (3) implica (1). ■

Grazie alla proprietà (2) del lemma 17.11, verificare che un insieme di programmi è indipendente diventa un compito banale (Cfr. Esercizio 17.5-2).

Il problema di minimizzare la somma delle penalità dei programmi in ritardo è esattamente uguale al problema di massimizzare la somma delle penalità dei programmi in anticipo. Pertanto, il teorema seguente assicura che possiamo utilizzare un algoritmo greedy per trovare un insieme di programmi A indipendenti e con la massima penalità.

Teorema 17.12

Sia S un insieme di n programmi di durata unitaria con scadenze, e sia $\mathcal{I}$ l'insieme di tutti gli insiemi indipendenti. Allora la struttura $(S, \mathcal{I})$ è un matroide.

Dimostrazione. Ogni sottoinsieme di un insieme di programmi indipendenti è sicuramente indipendente. Per dimostrare che vale la proprietà di scambio, supponiamo che A e B siano due insiemi di programmi indipendenti e che $|B| > |A|$. Sia k il più grande intero t per cui vale che $N_t(B) \le N_t(A)$. Poiché $N_n(B) = |B|$, $N_n(A) = |A|$ e $|B| > |A|$, abbiamo necessariamente che $k < n$, e che $N_j(B) > N_j(A)$ per tutti i valori di j nell'intervallo $k + 1 \le j \le n$. Pertanto, B contiene

	Problema						
	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 *Un esempio del problema dell'ordinamento su di un unico processore di programmi di durata unitaria con scadenze e penalità.*

un maggior numero di programmi con scadenza $k + 1$ di quanti ne contenga A. Sia x un programma appartenente all'insieme $B - A$ con scadenza $k + 1$. Poniamo $A' = A \cup \{x\}$.

A questo punto dimostriamo che A' è indipendente dimostrando che soddisfa la proprietà (2) del Lemma 17.11. Per $1 \le t \le k$, abbiamo che $N_t(A') = N_t(A) \le t$, dato che A è indipendente. Per $k < t \le n$, poiché B è indipendente vale che $N_t(A') \le N_t(B) \le t$. Ciò ci permette di concludere che A' è indipendente. Pertanto, la coppia $(S, \mathcal{I})$ è un matroide. ■

Il Teorema 17.10 assicura che un algoritmo greedy è in grado di trovare un insieme indipendente A di programmi con massimo peso. L'algoritmo costruisce una sequenza di esecuzione (o ordinamento) ottima in cui i programmi che appartengono a A sono in anticipo. Questa strategia ci permette di ottenere un algoritmo efficiente per definire l'ordinamento su un unico processore di programmi di durata unitaria con scadenze e penalità. Il tempo di esecuzione della procedura GREEDY è $O(n^2)$, poiché ognuno degli $O(n)$ controlli di indipendenza eseguiti dall'algoritmo necessita di $O(n)$ tempo (Cfr. Esercizio 17.5-2). Una realizzazione più efficiente è descritta nel Problema 17-3.

La figura 17.7 illustra un esempio del problema dell'ordinamento su un unico processore di programmi di durata unitaria con scadenze e penalità. In questo esempio l'algoritmo greedy seleziona i programmi 1, 2, 3 e 4, tralascia i programmi 5 e 6, ed infine accetta il programma 7. L'ordinamento ottimo risulta

$\langle 2, 4, 1, 3, 7, 5, 6 \rangle$.

Il totale delle penalità della sequenza risulta $w_5 + w_6 = 50$.

Esercizi

17.5-1 Si risolva il problema descritto nella figura 17.7, ma dove ogni penalità w_i è sostituita dalla penalità $80 - w_i$.

17.5-2 Si mostri come sia possibile utilizzare la proprietà 2 del Lemma 17.11 per determinare se un insieme A di programmi è indipendente in tempo $O(|A|)$.

Problemi

17-1 Il resto con monete

Consideriamo il problema di dare un resto di n lire con il minor numero possibile di monete.

a. Si descriva un algoritmo greedy che dia un resto di n lire usando monete del valore di lire 200, 100, 50 e 10. Si dimostri che l'algoritmo trova una soluzione ottima.

b. Si supponga che le monete disponibili siano $c^0, c^1, \ldots, c^k$ per un qualche intero $c > 1$ e $k \ge 1$. Si dimostri che l'algoritmo greedy restituisce sempre una soluzione ottima.

c. Si fornisca un insieme di monete per cui l'algoritmo greedy non restituisce una soluzione ottima.

17-2 Sottografi aciclici

a. Sia $G = (V, E)$ un grafo non orientato. Si dimostri che la coppia $(E, \mathcal{I})$, dove $A \in \mathcal{I}$ se e solo se A è un sottoinsieme di E senza cicli, è un matroide.

b. La ***matrice di incidenza di un grafo non orientato*** $G = (V, E)$ è una matrice M di dimensioni $|V| \times |E|$ tale che $M_{ve} = 1$, se l'arco e incide sul vertice v, $M_{ve} = 0$, altrimenti. Si dimostri che un insieme di colonne è linearmente indipendente se e solo se il corrispondente insieme di archi è senza cicli. Successivamente, si utilizzi il risultato dell'Esercizio 17.4-2 per dare una dimostrazione alternativa alla parte (a) dell'esercizio.

c. Supponiamo di associare a ogni arco di un grafo non orientato $G = (V, E)$, un peso $w(e)$ non negativo. Si definisca un algoritmo efficiente per trovare un sottoinsieme di E senza cicli e di massimo peso.

d. Sia $G = (V, E)$ un grafo orientato, e sia $(E, \mathcal{I})$ definito in modo tale che $A \in \mathcal{I}$ se e solo se A non contiene cicli orientati. Si fornisca un esempio di grafo orientato per cui la struttura $(E, \mathcal{I})$ non forma un matroide. Si specifichi quale sia la proprietà dei matroidi che non è soddisfatta.

e. La ***matrice di incidenza di un grafo orientato*** $G = (V, E)$ è una matrice M di dimensioni $|V| \times |E|$ tale che $M_{ve} = -1$, se l'arco e parte dal vertice v; $M_{ve} = 1$, se l'arco e arriva nel vertice v; $M_{ve} = 0$, altrimenti. Si dimostri che se un insieme di colonne è linearmente indipendente, allora il corrispondente insieme di archi non contiene un ciclo orientato.

f. L'Esercizio 17.4-2 afferma che l'insieme degli insiemi di colonne linearmente indipendenti di una qualunque matrice M forma un matroide. Per quale motivo i risultati delle parti (d) e (e) non sono in contraddizione? Per quale motivo non esiste una corrispondenza perfetta tra la proprietà di essere senza cicli di un insieme di archi, e la proprietà per cui l'insieme di colonne della matrice di incidenza associata è linearmente indipendente?

17-3 Variazioni del problema di scheduling

Il seguente algoritmo risolve il problema, descritto nel paragrafo 17.5, dell'ordinamento su un unico processore di programmi di durata unitaria con scadenze e penalità. Consideriamo un insieme n di intervalli consecutivi di tempo, dove l'intervallo i è quella quantità di tempo di lunghezza unitaria che termina al tempo i. Supponiamo che inizialmente gli n intervalli non siano associati a nessun programma. Supponiamo inoltre che i programmi siano ordinati per penalità decrescenti. L'algoritmo si comporta nel modo seguente: quando considera il programma j, se esistono degli intervalli di tempo che terminano prima o esattamente al tempo d_j e non ancora assegnati a nessun programma, allora l'algoritmo assegna al programma j il più grande tra questi intervalli. Altrimenti si assegna al programma j il più grande tra gli intervalli ancora liberi.

a. Si spieghi il motivo per cui l'algoritmo restituisce sempre una soluzione ottima.

b. Per realizzare in modo efficiente l'algoritmo si consiglia di utilizzare la foresta degli insiemi disgiunti descritta nel paragrafo 22.3. Si supponga che l'insieme dei programmi

in ingresso sia già ordinato per penalità decrescenti. Si analizzi il tempo di esecuzione della soluzione proposta.

Note al capitolo

Eccellenti riferimenti per i matroidi e gli algoritmi greedy sono Lawler [132] e Papadimitriou e Steiglitz [154].

Nella letteratura sull'ottimizzazione combinatoria, i primi riferimenti agli algoritmi greedy risalgono al lavoro di Edmonds [62] del 1971, anche se la teoria dei matroidi si può far risalire al lavoro di Whitney [200] del 1935.

La dimostrazione della correttezza dell'algoritmo greedy per il problema della selezione delle attività è basata su quella di Gavril [80]. Il problema dell'ordinamento di programmi è stato studiato da Lawler [132], Horowitz e Sahni [105] e Brassard e Bratley [33].

I codici di Huffman sono stati proposti nel 1952 [107]; una rassegna delle tecniche per la compressione dei dati proposte fino al 1987 si può trovare in Lelewer e Hirschberg [136].

Una estensione della teoria dei matroidi detta teoria dei "greedoidi" è presentata da Korte e Lovász [127, 128, 129, 130]. Questa teoria generalizza la teoria presentata in questo capitolo.

Analisi ammortizzata

18

Nell'*analisi ammortizzata* il tempo di esecuzione di una sequenza di operazioni di una struttura di dati è definito come la media dei tempi delle operazioni della sequenza. Generalmente, l'analisi ammortizzata è utilizzata per dimostrare che il costo di una certa trasformazione è piccolo, se lo si considera come costo medio di una sequenza di operazioni, anche nel caso in cui alcune operazioni della trasformazione siano piuttosto costose. L'analisi ammortizzata differisce dall'analisi del caso medio per il fatto che non si fa riferimento a concetti della teoria delle probabilità: una analisi ammortizzata considera l'*efficienza media di ogni operazione nel caso pessimo*.

I primi tre paragrafi di questo capitolo esaminano le tre tecniche comunemente usate per l'analisi ammortizzata. Il paragrafo 18.1 prende in esame il metodo degli aggregati. In questa tecnica prima si determina un limite superiore $T(n)$ del costo totale di una sequenza di n operazioni. Il costo ammortizzato di una operazione è dato da $T(n)/n$.

Il paragrafo 18.2 esamina il metodo degli accantonamenti, in cui si stabilisce il costo ammortizzato di ogni operazione. Quando si considerano operazioni di tipo diverso, allora può accadere che il costo ammortizzato di una operazione vari a seconda del tipo. Il metodo degli accantonamenti addebita un costo aggiuntivo a determinate operazioni della sequenza: questo addebito deve essere considerato come un "credito prepagato" relativamente a quello specifico oggetto della struttura di dati. Successivamente, il credito è usato per definire il costo di quelle operazioni a cui altrimenti verrebbe addebitato un costo minore di quello reale.

Il paragrafo 18.3 esamina il metodo del potenziale. Questo metodo ha caratteristiche simili al metodo degli accantonamenti, nel senso che si stabilisce il costo ammortizzato di ogni operazione, sovrastimando, inizialmente, il costo di determinate operazioni; successivamente, questo addebito viene utilizzato per compensare i costi sottostimati. Differentemente dal metodo degli aggregati, dove il credito è associato agli oggetti individuali della struttura di dati, nel metodo del potenziale il credito è più propriamente visto come "l'energia potenziale", o più semplicemente "il potenziale", della struttura di dati.

Nel seguito, per descrivere le caratteristiche di questi tre modelli prenderemo in esame due esempi concreti. Un primo esempio è dato dalla struttura di dati pila, estesa con una ulteriore operazione: l'operazione MULTIPOP. L'operazione MULTIPOP elimina dalla pila un certo numero di oggetti. Il secondo esempio è dato da un contatore binario che conta, a partire da 0, con un'unica operazione INCREMENT.

Durante la lettura di questo capitolo, si deve sempre tenere a mente che i costi assegnati durante una analisi ammortizzata sono utilizzati solamente per effettuare l'analisi; non sono

presenti nel programma reale. Per esempio, se nel caso del metodo degli accantonamenti si assegna un certo credito a un oggetto x, non c'è alcuna necessità di assegnare lo stesso valore a qualche attributo *credit*[x] del codice.

Le informazioni su una particolare struttura di dati, che si ottengono a seguito di una analisi ammortizzata, possono essere utilizzate per ottimizzare il progetto di algoritmi. Per esempio, nel paragrafo 18.4 utilizzeremo il metodo del potenziale per analizzare le caratteristiche di una tabella che, dinamicamente, modifica le proprie dimensioni.

18.1 Il metodo degli aggregati

Nel ***metodo degli aggregati*** occorre mostrare che, per tutti i valori di n, una sequenza di n operazioni richiede, nel *caso pessimo*, tempo $T(n)$.

Quindi, nel caso pessimo, il costo medio, o ***costo ammortizzato***, di una operazione è dato da $T(n)/n$. Si noti che questo costo ammortizzato viene attribuito alla singola operazione e si applica anche al caso in cui nella sequenza siano presenti operazioni di tipo diverso.

Gli altri due metodi che esamineremo in questo capitolo, il metodo degli accantonamenti ed il metodo del potenziale, possono assegnare un costo ammortizzato diverso a operazioni di tipo diverso.

Operazioni su pile

Nel nostro primo esempio di analisi con il metodo degli aggregati, prenderemo in esame la struttura di dati pila, estesa con una nuova operazione. Nel paragrafo 11.1 sono state presentate le operazioni fondamentali della struttura di dati pila. Queste operazioni richiedono tempo $O(1)$:

PUSH(S, x) inserisce l'oggetto x in testa alla pila S.

POP(S) toglie l'elemento in testa alla pila S e restituisce quell'elemento.

Dato che queste operazioni vengono eseguite in tempo $O(1)$, possiamo considerare che abbiano un costo unitario. Quindi, il costo totale di una sequenza di n operazioni PUSH e POP è n e il tempo di esecuzione reale di una sequenza di n operazioni risulta $\Theta(n)$.

L'analisi ammortizzata diventa più interessante se estendiamo la struttura di dati pila con una nuova operazione: l'operazione MULTIPOP. L'operazione MULTIPOP(S, k) rimuove i primi k elementi dalla testa della pila, oppure svuota la pila nel caso in cui la pila S contenga un numero di elementi minore di k. Nella procedura seguente, l'operazione STACK-EMPTY restituisce TRUE se la pila non contiene elementi, altrimenti restituisce FALSE.

```
MULTIPOP(S, k)
1   while STACK-EMPTY(S) ≠ TRUE e k ≠ 0
2       do POP(S)
3          k ← k − 1
```

Figura 18.1 *L'effetto dell'esecuzione di una operazione* MULTIPOP *su una pila S, la cui struttura iniziale è descritta in* (a). *L'operazione* MULTIPOP(S, 4) *elimina quattro oggetti dalla testa della pila, il risultato dell'operazione è mostrato in* (b). *La successiva operazione* MULTIPOP(S, 7) *svuota la pila – il risultato dell'operazione è mostrato in* (c) *– dato che la pila contiene un numero di elementi minori di 7.*

La figura 18.1 mostra un esempio di applicazione dell'operazione MULTIPOP.

A questo punto ci domandiamo quale sia il tempo di esecuzione di MULTIPOP(S, k) applicata a una pila S con s elementi. Il tempo di esecuzione è lineare nel numero di operazioni POP effettivamente eseguite: questo è sufficiente per dire che il costo della operazione MULTIPOP può essere studiato considerando il costo unitario astratto delle operazioni di POP e PUSH. Il numero di iterazioni del ciclo **while** è dato dal numero min(s, k) di elementi che sono eliminati dalla testa della pila. Inoltre, durante ogni iterazione del ciclo si esegue una sola chiamata dell'operazione POP (linea 2). In conclusione, il costo totale dell'operazione MULTIPOP è dato da min(s, k) e il suo tempo di esecuzione effettivo è una funzione lineare di questo costo.

Consideriamo una sequenza di n operazioni POP, PUSH e MULTIPOP su una pila inizialmente vuota. Nel caso pessimo il costo di una operazione MULTIPOP nella sequenza è $O(n)$, dato che la dimensione della pila sarà al massimo n. Pertanto, il caso pessimo del costo di una singola operazione è dato da $O(n)$ e questo implica che il costo di una sequenza di n operazioni risulta $O(n^2)$: infatti possiamo avere $O(n)$ operazioni MULTIPOP il cui costo è $O(n)$ per ciascuna. Sebbene questa analisi sia corretta, il risultato $O(n^2)$, ottenuto considerando il costo pessimo delle singole operazioni, non è accurato.

Usando il metodo degli aggregati dell'analisi ammortizzata possiamo ottenere un limite superiore più significativo considerando l'intera sequenza delle n operazioni. Infatti, anche se una operazione MULTIPOP può essere piuttosto costosa, una sequenza qualsiasi di n operazioni POP, PUSH e MULTIPOP su una pila inizialmente vuota può costare al massimo $O(n)$. Qual è la ragione di questa proprietà? Ogni elemento può essere tolto dalla pila solo dopo esservi stato inserito con un'operazione di PUSH. Quindi, il numero delle chiamate dell'operazione POP, considerando anche quelle all'interno della operazione MULTIPOP, al massimo può coincidere con il numero di operazioni PUSH, e questo numero è, al massimo, n. Per ogni valore di n, una sequenza di n operazioni POP, PUSH e MULTIPOP richiede un tempo totale $O(n)$. Il costo ammortizzato di una operazione è dato dalla media $O(n)/n = O(1)$.

Vogliamo ancora una volta mettere in evidenza che non abbiamo fatto alcun riferimento a concetti di probabilità, anche se abbiamo mostrato che il costo medio, e quindi il tempo di esecuzione, di un'operazione della struttura di dati pila è $O(1)$. In realtà abbiamo dimostrato che $O(n)$ è un limite superiore per il *caso pessimo* di una sequenza di n operazioni. Dividendo questo limite per il numero delle operazioni della sequenza si ottiene il costo medio per operazione, cioè il costo ammortizzato.

Incremento di un contatore binario

Un altro esempio di applicazione del metodo degli aggregati è dato dal problema di realizzare un contatore binario, con k bit, che conta a partire da 0. Il contatore è realizzato da un array $A[0 \,..\, k-1]$ di bit, dove $length[A] = k$. Il bit meno significativo del numero binario x memorizzato nel contatore è rappresentato dall'elemento $A[0]$, mentre il bit più significativo è $A[k-1]$.

Pertanto $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$.

Inizialmente, $x = 0$, e, quindi, $A[i] = 0$ per $i = 0, 1, \ldots, k-1$. Per sommare 1 (modulo 2^k) al valore corrente del contatore facciamo uso della procedura seguente.

```
INCREMENT(A)
1  i ← 0
2  while i < length[A] e A[i] = 1
3        do  A[i] ← 0
4            i ← i + 1
5  if i < length[A]
6        then A[i] ← 1
```

Questo algoritmo è fondamentalmente quello realizzato hardware da un addizionatore con propagazione del riporto (paragrafo 29.2.1). La figura 18.2 mostra cosa accade al contatore quando viene incrementato per 16 volte: il valore iniziale è 0 mentre il valore finale è 16. All'inizio di ogni iterazione del ciclo **while** (linee 2-4) si vuole sommare 1 nella posizione i. Se $A[i] = 1$ allora come risultato della somma abbiamo che il bit nella posizione i diviene 0, e si deve riportare 1 da sommare alla posizione $i + 1$ nella successiva iterazione del ciclo. Nel caso contrario, il ciclo termina e allora, se $i < k$, sicuramente $A[i] = 0$ e si deve semplicemente fare in modo che il bit in posizione i diventi 1 (linea 6). Il costo di una operazione INCREMENT è lineare nel numero di operazioni di modifica dei bit.

Come nel caso della pila, una analisi puntuale fornisce un limite corretto ma poco significativo. Una singola esecuzione dell'operazione INCREMENT richiede, nel caso pessimo dove tutti gli elementi dell'array sono uguali a 1, un tempo $\Theta(k)$. Quindi, una sequenza di n operazioni INCREMENT su di un contatore, inizialmente posto a zero, richiede un tempo $O(nk)$ nel caso pessimo.

Possiamo rendere la nostra analisi più precisa notando che non tutti i bit vengono complementati a ogni chiamata dell'operazione INCREMENT. In questo modo il costo del caso pessimo di una sequenza di n operazioni INCREMENT diventa $O(n)$. Come mostrato nella figura 18.2, $A[0]$ viene complementato a ogni chiamata dell'operazione INCREMENT. Invece, in una sequenza di n operazioni INCREMENT su di un contatore inizialmente posto a zero, il secondo bit più significativo $A[1]$, viene complementato $\lfloor n/2 \rfloor$ volte. In modo analogo si dimostra che, in una sequenza di n operazioni INCREMENT, $A[2]$ viene complementato $\lfloor n/4 \rfloor$ volte. In generale, per $i = 0, 1, \ldots, \lfloor \lg n \rfloor$, in una sequenza di n operazioni INCREMENT su di un contatore inizialmente posto a zero, il bit $A[i]$ viene complementato $\lfloor n/2^i \rfloor$ volte, mentre per $i > \lfloor \lg n \rfloor$, il bit $A[i]$ non viene complementato affatto. Il numero totale di operazioni di modifica di bit, grazie all'equazione (3.4), diviene

Valore del contatore	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Costo totale
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 *Le trasformazioni del valore di un contatore binario a 8 bit in una sequenza di 16 operazioni* INCREMENT. *I bit rappresentati all'interno di rettangoli grigi sono quelli che vengono complementati per ottenere il valore successivo del contatore. Il costo dell'operazione è riportato nella parte destra della figura. Si noti che il costo totale non è mai maggiore del doppio del numero totale di operazioni* INCREMENT.

$$\begin{aligned}\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n \,.\end{aligned}$$

In conclusione, il tempo richiesto nel caso pessimo da una sequenza di n operazioni INCREMENT su di un contatore inizialmente posto a zero è $O(n)$, quindi il costo ammortizzato di una singola operazione risulta $O(n)/n = O(1)$.

Esercizi

18.1-1 Supponiamo di estendere la struttura di dati pila con una operazione MULTIPUSH. Anche in questo caso il limite del costo ammortizzato delle operazioni della struttura di dati pila continua a essere $O(1)$?

18.1-2 Si dimostri che se la struttura del contatore binario di k bit fosse estesa con una operazione DECREMENT, allora il costo di n operazioni diventerebbe $\Theta(nk)$ nel caso peggiore.

18.1-3 Supponiamo di eseguire una sequenza di n operazioni su una determinata struttura di dati. Supponiamo inoltre che la i-esima operazione abbia costo i se i è una potenza di 2, altrimenti il suo costo sia 1. Si utilizzi il metodo degli aggregati per stabilire il costo ammortizzato.

18.2 Il metodo degli accantonamenti

Il ***metodo degli accantonamenti*** dell'analisi ammortizzata assegna un valore diverso alle diverse operazioni. Può accadere che in questo modo una determinata operazione abbia un valore maggiore o minore del suo costo effettivo. Il valore assegnato alle operazioni è chiamato ***costo ammortizzato***. Nel caso in cui il costo ammortizzato di una operazione è maggiore del costo effettivo, allora la differenza associata a un determinato oggetto della struttura di dati viene detta ***credito***. Il credito viene usato successivamente per compensare quelle operazioni i cui costi ammortizzati risultano minori dei costi effettivi. Il costo ammortizzato di una operazione può essere visto come composto dal costo effettivo e dal credito, che può essere sia depositato che prelevato. In questa ottica, il metodo degli accantonamenti presenta numerose differenze rispetto al metodo degli aggregati in cui tutte le operazioni hanno lo stesso costo ammortizzato.

La scelta del costo ammortizzato di una operazione deve essere oculata. Se l'obiettivo è dimostrare che, con una opportuna scelta dei costi ammortizzati, nel caso pessimo il costo medio per operazione è piccolo, allora il costo ammortizzato totale di una sequenza di operazioni deve essere un limite superiore del costo effettivo totale della sequenza. Inoltre, come nel caso del metodo degli aggregati, questa proprietà deve valere per tutte le sequenze di operazioni. Quindi, il credito totale associato a una determinata struttura di dati deve essere sempre non negativo, poiché deve rappresentare sempre (per ogni sequenza) quel valore di cui il costo ammortizzato totale supera il costo effettivo totale. Infatti, se fosse possibile avere un credito totale negativo (questo sarebbe il risultato di un precedente addebito di una operazione sottostimata con una promessa di un successivo pagamento del debito), allora i possibili totali dei costi ammortizzati diventerebbero inferiori ai possibili totali dei costi effettivi. Pertanto, il costo ammortizzato totale non sarebbe un limite superiore del costo effettivo totale. Perciò, il credito totale associato a una struttura di dati non deve mai diventare minore di zero.

Operazioni su pile

Per illustrare le caratteristiche del metodo degli accantonamenti, consideriamo nuovamente l'esempio della pila. Prima di tutto ricordiamo i costi effettivi delle operazioni:

PUSH 1,
POP 1,
MULTIPOP $\min(k, s)$,

dove k è il numero degli elementi da rimuovere nell'operazione MULTIPOP, e s rappresenta la dimensione della pila al momento della chiamata dell'operazione. Supponiamo di assegnare alle operazioni i seguenti costi ammortizzati:

PUSH 2,
POP 0,
MULTIPOP 0.

Si osservi che il costo ammortizzato dell'operazione MULTIPOP è una costante (0), mentre il costo effettivo è una quantità variabile. In questo caso, i costi ammortizzati sono tutti $O(1)$ ma, in generale, può accadere che i costi ammortizzati delle operazioni in esame differiscano tra loro asintoticamente.

Andiamo a questo punto a vedere quanto viene a costare una qualunque sequenza di operazioni su pile addebitando i costi ammortizzati. Supponiamo di effettuare i pagamenti in dollari e che, inizialmente, la pila sia vuota. Si ricordi l'analogia, presentata nel paragrafo 11.1, tra la struttura di dati pila e una pila di piatti di un ristorante. Quando si mette un nuovo piatto sulla pila, il costo effettivo dell'operazione è 1 dollaro, e si rimane con un credito di un dollaro (dei due dollari addebitati come costo ammortizzato dell'operazione). Questo credito di un dollaro è messo sopra il piatto. Pertanto, a ogni istante di tempo, tutti i piatti presenti nella pila possiedono un dollaro di credito.

Il dollaro messo sul piatto può essere visto come un pagamento anticipato del costo dell'operazione che rimuove il piatto dalla pila. Nel momento in cui si esegue una operazione di POP, non si addebita costo alcuno, ma si paga il costo effettivo con il credito accantonato nella pila. Quindi, per rimuovere un piatto, si deve prima prendere il dollaro di credito per poterlo successivamente utilizzare per pagare il costo effettivo dell'operazione. In questo modo, un addebito leggermente superiore dell'operazione di PUSH, permette di non aver bisogno di addebitare nessun costo per l'operazione di POP.

Inoltre, non abbiamo bisogno di addebitare nessun costo neanche per l'operazione MULTIPOP. Infatti, per rimuovere il primo piatto basta utilizzare il suo credito per pagare il costo effettivo dell'operazione POP. Analogamente, per rimuovere il secondo piatto continuiamo a utilizzare il credito per pagare la successiva operazione POP e così via. Quindi, abbiamo sempre un credito sufficiente per pagare l'operazione MULTIPOP. In altri termini, visto che per ogni piatto sulla pila si ha un dollaro di credito e che la pila contiene sempre un numero di piatti maggiore o uguale a zero, siamo sicuri che il credito totale è sempre maggiore o uguale a zero. Pertanto, per *ogni* sequenza di n operazioni PUSH, POP e MULTIPOP, il costo ammortizzato totale è un limite superiore del costo effettivo totale. Il costo ammortizzato totale è $O(n)$, così come il costo effettivo totale.

Incremento di un contatore binario

Possiamo fornire un altro esempio di applicazione del metodo degli accantonamenti analizzando l'operazione INCREMENT su di un contatore binario inizializzato a zero. Come abbiamo già osservato in precedenza, il tempo di esecuzione di questa operazione è proporzionale al numero delle operazioni di modifica di singoli bit. Questo numero definisce la nostra nozione di costo. Ancora una volta supponiamo che i costi siano pagati in dollari.

Prima di tutto, addebitiamo un costo ammortizzato di due dollari per l'operazione che pone un bit a 1. Quando viene richiesto di mettere un bit a 1, utilizziamo un dollaro (dei due dollari addebitati) per pagare il costo effettivo di questa operazione. Il dollaro rimanente rappresenta il credito associato al bit. A ogni istante di tempo, ogni 1 del contatore ha associato un credito di un dollaro. Dunque non abbiamo bisogno di addebitare nessun costo all'operazione che riporta il bit a 0: per pagare questa operazione ci basta prendere il dollaro di credito associato al bit.

A questo punto l'analisi ammortizzata dell'operazione INCREMENT è presto fatta. Il costo dell'operazione che riporta a 0 i bit all'interno del ciclo **while** viene pagato con i dollari

associati ai bit. Dato che l'operazione INCREMENT al più porta a 1 un solo bit (linea 6), allora il costo ammortizzato dell'operazione risulta (al più) di 2 dollari. Inoltre, il numero totale di 1 presenti nel contatore non è mai negativo e, quindi, il totale del credito è sempre maggiore o uguale a zero. In conclusione, una sequenza di n operazioni INCREMENT ha un costo ammortizzato totale di $O(n)$, che è un limite superiore del costo effettivo totale.

Esercizi

18.2-1 Supponiamo di eseguire una sequenza di operazioni su di una pila in modo tale che la dimensione della pila non sia mai maggiore di k. Dopo k operazioni viene sempre fatta una copia della pila. Si dimostri che, assegnando un costo ammortizzato appropriato alle diverse operazioni, il costo di n operazioni sulla pila è $O(n)$, incluse le operazioni di copia.

18.2-2 Ripetere l'Esercizio 18.1-3 con il metodo di analisi ammortizzata degli accantonamenti.

18.2-3 Supponiamo di voler definire, oltre all'operazione che incrementa un contatore, anche una operazione che mette a zero tutti i bit del contatore (operazione RESET). Si mostri come realizzare un contatore con un vettore di bit in modo tale che una sequenza di n operazioni INCREMENT e RESET su di un contatore inizializzato a zero venga eseguita in tempo $O(n)$. (*Suggerimento*: si può usare un puntatore alla posizione della cifra 1 più significativa del numero binario rappresentato dal contatore).

18.3 Il metodo del potenziale

Invece di rappresentare il lavoro pagato in anticipo come credito associato con un particolare oggetto della struttura di dati, il metodo di analisi ammortizzata conosciuto con il nome di ***metodo del potenziale*** rappresenta il lavoro pagato in anticipo come "l'energia potenziale", o più semplicemente "il potenziale", che deve essere utilizzato per pagare le operazioni future. Il potenziale è associato alla struttura di dati nella sua interezza e non a un particolare oggetto della struttura di dati.

Il metodo del potenziale è definito nel modo seguente. Prima di tutto si considera una struttura di dati iniziale D_0; le n operazioni verranno eseguite su di essa. Per ogni $i = 1, 2, \ldots, n$, indichiamo con c_i il costo effettivo dell'operazione i-esima e indichiamo con D_i la struttura di dati che si ottiene come risultato dell'applicazione della operazione i alla struttura di dati D_{i-1}. Una ***funzione potenziale*** Φ è una funzione che associa un numero reale $\Phi(D_i)$ alla struttura di dati D_i; il numero reale $\Phi(D_i)$ è detto il ***potenziale*** associato alla struttura di dati D_i. Il ***costo ammortizzato*** $\hat{c}_i$ della i-esima operazione rispetto alla funzione di potenziale Φ è così definito:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \qquad (18.1)$$

Il costo ammortizzato di ogni operazione è dunque dato dalla somma del costo effettivo con l'aumento di potenziale dovuto all'operazione. Il costo ammortizzato totale di n operazioni diventa

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) . \qquad (18.2)\end{aligned}$$

La prima uguaglianza è una diretta conseguenza dell'equazione (18.1), mentre la seconda uguaglianza si ricava dall'equazione (3.7).

Per una funzione potenziale Φ con $\Phi(D_n) \geq \Phi(D_0)$, il costo ammortizzato totale $\sum_{i=1}^{n} \hat{c}_i$ è un limite superiore del costo effettivo totale. In realtà, il numero totale di operazioni che potrebbero essere eseguite non è sempre noto; pertanto, richiedere che sia $\Phi(D_i) \geq \Phi(D_0)$, per tutti i valori di i, significa garantire, come nel caso del metodo degli accantonamenti, dei pagamenti anticipati. Spesso, conviene definire la funzione potenziale in modo tale che $\Phi(D_0) = 0$, e poi dimostrare che $\Phi(D_i) \geq 0$ per tutti i valori di i. (L'Esercizio 18.3-1 presenta un metodo semplice che permette di trattare il caso in cui $\Phi(D_0) \neq 0$.)

Una differenza di potenziale $\Phi(D_i) - \Phi(D_{i-1})$ positiva della i-esima operazione ha il significato intuitivo di un versamento: il costo ammortizzato $\hat{c}_i$ descrive un versamento associato all'operazione i-esima che fa aumentare il potenziale della struttura di dati. Invece, se la differenza di potenziale è negativa, il costo ammortizzato ha il significato di un prelievo effettuato dalla operazione i-esima: il costo effettivo dell'operazione è pagato con una diminuzione del potenziale.

I costi ammortizzati definiti dalle due equazioni (18.1) e (18.2) dipendono dalla scelta della funzione potenziale Φ. Funzioni potenziale differenti possono dare origine a costi ammortizzati differenti, purché risultino sempre limiti superiori dei costi effettivi. Spesso, la scelta della funzione potenziale è il risultato di un compromesso: quale sia la migliore funzione potenziale dipende dai limiti di tempo desiderati.

Operazioni su pile

Per illustrare le caratteristiche del metodo del potenziale, consideriamo nuovamente l'esempio delle operazioni della struttura di dati pila con le operazioni PUSH, POP e MULTIPOP. La funzione potenziale Φ della struttura di dati pila è definita dal numero degli elementi della pila. Nel caso della pila vuota D_0, la struttura di dati iniziale, abbiamo che $\Phi(D_0) = 0$. Dato che il numero degli elementi di una pila non è mai minore di zero, la pila D_i che si ottiene come risultato dell'esecuzione della i-esima operazione ha un potenziale maggiore o uguale a zero. Dunque

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0) .\end{aligned}$$

Il costo ammortizzato totale di n operazioni rispetto alla funzione potenziale Φ rappresenta un limite superiore del costo effettivo.

A questo punto calcoliamo i costi ammortizzati delle diverse operazioni della struttura di dati pila. Se la i-esima operazione è una operazione PUSH e la pila contiene s elementi, allora la differenza di potenziale risulta

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 .\end{aligned}$$

Il costo ammortizzato della operazione PUSH diventa

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 .\end{aligned}$$

Si supponga che la i-esima operazione sia l'operazione MULTIPOP(S, k) e che $k' = \min(k, s)$ sia il numero degli elementi tolti dalla pila. Abbiamo che il costo effettivo dell'operazione è k', mentre la differenza di potenziale è

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' .$$

Dunque, il costo ammortizzato dell'operazione MULTIPOP è

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 .\end{aligned}$$

In modo analogo si dimostra che il costo ammortizzato di una operazione POP è 0.

Il costo ammortizzato di tutte e tre le operazioni è $O(1)$, e, quindi, il costo ammortizzato totale di una sequenza di n operazioni è $O(n)$. Abbiamo fatto notare in precedenza che $\Phi(D_i) \geq \Phi(D_0)$; da questo deriva che il costo ammortizzato totale di n operazioni è un limite superiore del costo effettivo totale. Il costo di n operazioni nel caso pessimo è pertanto $O(n)$.

Incremento di un contatore binario

Come altro esempio di applicazione del metodo del potenziale consideriamo nuovamente l'esempio dell'incremento di un contatore binario. In questo caso, il potenziale del contatore dopo l'esecuzione della i-esima operazione INCREMENT è definito dal valore b_i, il numero di bit uguali a 1 presenti nel contatore dopo l'esecuzione della i-esima operazione.

Calcoliamo ora il costo ammortizzato della operazione INCREMENT. Supponiamo che la i-esima operazione INCREMENT porti a zero t_i bit. Il costo effettivo dell'operazione è al più $t_i + 1$, dato che, oltre a mettere a zero t_i bit, l'operazione può al più mettere un altro bit a 1. Il numero dei bit uguali a 1 nel contatore dopo l'esecuzione della i-esima operazione è dunque $b_i \leq b_{i-1} - t_i + 1$. Da questo deriva che la differenza di potenziale è

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i .\end{aligned}$$

Infine, il costo ammortizzato risulta

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 .\end{aligned}$$

Se il contatore binario è inizializzato a zero, allora $\Phi(D_0) = 0$. Dato che per tutti i valori di i, $\Phi(D_i) \geq 0$, il costo ammortizzato totale di una sequenza di n operazioni INCREMENT è un limite superiore del costo effettivo totale. Pertanto, il costo di n operazioni INCREMENT nel caso pessimo è $O(n)$.

Il metodo del potenziale fornisce un meccanismo molto semplice per analizzare il contatore anche nel caso in cui il suo valore iniziale non sia uguale a zero. Supponiamo che inizialmente il contatore contenga un numero b_0 di bit uguali a 1, e che dopo n operazioni INCREMENT ci siano b_n bit uguali a 1, dove $0 \leq b_0, b_n \leq k$. Possiamo riscrivere l'equazione (18.2) nel modo seguente:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) . \tag{18.3}$$

Poiché $\hat{c}_i \leq 2$, per $1 \leq i \leq n$, $\Phi(D_0) = b_0$ e $\Phi(D_n) = b_n$, il costo effettivo totale di n operazioni INCREMENT è

$$\begin{aligned}\sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 .\end{aligned}$$

In particolare si noti che $b_0 \leq k$ e quindi l'esecuzione di almeno $n = \Omega(k)$ operazioni INCREMENT ha un costo effettivo totale uguale a $O(n)$, indipendentemente da quale sia il valore iniziale del contatore.

Esercizi

18.3-1 Supponiamo di considerare una funzione potenziale Φ tale che $\Phi(D_i) \geq \Phi(D_0)$, per tutti i valori di i, ma che $\Phi(D_0) \neq 0$. Si dimostri che esiste una funzione potenziale Φ' tale che $\Phi'(D_0) = 0$ e $\Phi'(D_i) \geq 0$, per $i \geq 1$, ed inoltre che i costi ammortizzati rispetto alla funzione Φ' sono gli stessi costi ammortizzati della funzione Φ.

18.3-2 Risolvere nuovamente l'Esercizio 18.1-3 con il metodo del potenziale.

18.3-3 Si consideri una generica struttura di heap binario con n elementi, tale che le operazioni INSERT e EXTRACT-MIN siano eseguite in tempo $O(\lg n)$ nel caso pessimo. Si definisca una funzione potenziale Φ tale che il costo ammortizzato dell'operazione INSERT sia $O(\lg n)$, e il costo ammortizzato dell'operazione EXTRACT-MIN sia $O(1)$. Si dimostri che la funzione potenziale è ben definita.

18.3-4 Qual è il costo totale dell'esecuzione di n operazioni di PUSH, POP e MULTIPOP nel caso in cui inizialmente la pila contenga s_0 elementi e al termine della sequenza di operazioni la pila contenga s_n elementi?

18.3-5 Supponiamo che il valore iniziale del contatore binario sia un numero diverso da zero. Sia b il numero di 1 della rappresentazione binaria del valore iniziale del contatore. Si dimostri che il costo dell'esecuzione di n operazioni INCREMENT è $O(n)$, se $n = \Omega(b)$. (Non si assuma che b sia costante.)

18.3-6 Si mostri come sia possibile realizzare una coda utilizzando due pile (cfr. Esercizio 11.1.6) in modo tale che il costo ammortizzato delle operazioni ENQUEUE e DEQUEUE sia $O(1)$.

18.4 Tabelle dinamiche

In molte classi di applicazioni non è possibile sapere a priori quale sia il numero di elementi che dovranno essere memorizzati in una tabella. Potrebbe accadere di assegnare un certo spazio di memoria a una tabella per accorgersi successivamente che lo spazio assegnato non è più sufficiente. Se questo accade, la tabella deve essere spostata in una zona di memoria più grande, e tutti gli elementi memorizzati nella tabella originaria devono essere ricopiati nella tabella più grande. Analogamente, può accadere che si elimini un grande numero di elementi della tabella, allora potrebbe essere utile assegnare alla tabella una porzione di memoria più piccola. In questo paragrafo studiamo il problema dell'espansione e contrazione dinamica di tabelle. Utilizzando le tecniche dell'analisi ammortizzata dimostreremo che il costo ammortizzato delle operazioni di inserimento ed eliminazione (di un elemento) è $O(1)$, anche se il costo effettivo di una operazione è maggiore in presenza di una espansione o una contrazione. Inoltre, mostreremo come sia possibile assicurare che lo spazio non utilizzato di una tabella dinamica non superi mai una frazione costante dello spazio totale.

Assumiamo che le tabelle dinamiche forniscano le operazioni TABLE-INSERT e TABLE-DELETE. L'operazione TABLE-INSERT inserisce nella tabella un elemento che occupa una singola ***posizione***, dove una posizione rappresenta lo spazio necessario per memorizzare un singolo elemento. Analogamente, l'operazione TABLE-DELETE può essere pensata come quell'operazione che elimina un elemento dalla tabella ed in questo modo rende disponibile la posizione relativa. I dettagli delle tecniche di strutturazione dei dati adottate per organizzare la tabella non sono particolarmente importanti: potremmo utilizzare una pila (paragrafo 11.1), uno heap (paragrafo 7.1), una tabella hash (Capitolo 12). Per organizzare la memoria potremmo utilizzare un array, o un insieme di array, come descritto nel paragrafo 11.3.

È conveniente fare uso di un concetto introdotto nell'analisi dei metodi hash (Capitolo 12). Definiamo il ***fattore di carico*** $\alpha(T)$ di una tabella non vuota T, come il risultato della divisione tra il numero degli elementi memorizzati nella tabella e la dimensione (il numero delle posizioni) della tabella. Per definizione, la tabella vuota (quella senza elementi) ha dimensione 0 e il suo fattore di carico è 1. Se il fattore di carico di una tabella dinamica è limitato inferiormente da una costante, allora lo spazio non utilizzato nella tabella non è mai maggiore di una frazione costante dello spazio totale assegnato alla tabella.

Prima di tutto consideriamo una tabella dinamica dove sono possibili solo operazioni di inserzione. Successivamente, considereremo il caso più generale in cui sono permesse sia inserzioni che eliminazioni.

18.4.1 Espansione della tabella

Supponiamo che lo spazio assegnato alla tabella sia organizzato come un array di posizioni. Una tabella è piena quando sono state usate tutte le posizioni o, equivalentemente, quando il suo fattore di carico è diventato 1.[1] In alcuni ambienti software, ogni tentativo di inserzione di un elemento in una tabella piena dà sempre origine a una eccezione che segnala la presenza di un errore. In questo paragrafo, invece, assumiamo che il nostro ambiente software, come la maggior parte degli ambienti evoluti, sia equipaggiato con un sistema per la gestione della memoria. Il sistema di gestione della memoria è in grado di assegnare e liberare, se richiesto, blocchi di memoria. Pertanto, quando si deve inserire un elemento in un tabella piena, è possibile ***espandere*** la tabella. L'operazione di estensione prima crea una nuova tabella con un numero di posizioni maggiore di quella vecchia e, successivamente, copia tutti gli elementi della vecchia tabella nella nuova.

Una strategia euristica comunemente adottata per l'espansione della tabella consiste nel duplicare il numero di posizioni assegnate alla nuova tabella. Se si eseguono solamente operazioni di inserzione, allora il fattore di carico della tabella è sempre almeno 1/2, e pertanto la quantità totale di spazio sprecato non supera mai la metà dello spazio totale della tabella.

Nella seguente procedura indichiamo la tabella con T, *table*[T] indica il puntatore al blocco di memoria che contiene la tabella, *num*[T] contiene il numero degli elementi memorizzati nella tabella, e *size*[T] contiene il numero totale delle posizioni presenti nella tabella. Inizialmente la tabella è vuota: *num*[T] = *size*[T] = 0.

```
TABLE-INSERT(T, x)
  if size[T] = 0
        then  assegna spazio a table[T] con una sola posizione
              size[T] ← 1
  if num[T] = size[T]
        then  assegna spazio a new-table con 2 size[T] posizioni
              inserisci tutti gli elementi di table[T] in new-table
              libera lo spazio occupato da table[T]
              table[T] ← new-table
              size[T] ← 2 size[T]
  inserisci x in table[T]
  num[T] ← num[T] + 1
```

[1] In qualche caso, come quello di una tabella hash con indirizzamento aperto, può essere opportuno considerare piena una tabella quando il suo fattore di carico diventa uguale a una costante minore di uno (si veda l'Esercizio 18.4-2).

Si noti che possiamo distinguere due procedure di "inserzione": la procedura stessa TABLE-INSERT e la procedura di ***inserzione elementare*** in una tabella, utilizzata nelle linee 6 e 10. Possiamo analizzare il tempo di esecuzione della procedura TABLE-INSERT in termini del numero di inserzioni elementari associando un costo unitario a ogni operazione di inserzione elementare. Assumiamo che il tempo di esecuzione effettivo della procedura TABLE-INSERT sia lineare rispetto al tempo necessario per inserire un singolo elemento, così che il costo del trasferimento degli elementi (linea 6) domini sia il costo supplementare necessario per assegnare la memoria della tabella iniziale (linea 2), sia il costo supplementare per assegnare e liberare blocchi di memoria (linee 5 e 7). Chiamiamo ***espansione*** l'evento che corrisponde all'esecuzione delle operazioni che seguono la clausola **then** (linee 5-9).

A questo punto ci poniamo il problema di analizzare una sequenza di n operazioni TABLE-INSERT su di una tabella inizialmente vuota. Qual è il costo c_i della i-esima operazione? Se c'è spazio sufficiente nella tabella corrente (o se questa è la prima operazione), allora $c_i = 1$, dato che si deve eseguire una sola operazione di inserzione elementare alla linea 10. Se la tabella corrente è piena, tuttavia, diviene necessaria una espansione. In questo caso $c_i = i$: il costo è dato dalla somma della operazione di inserzione elementare (linea 10), con il costo $i - 1$ dell'operazione che copia tutti gli elementi della vecchia tabella nella nuova (linea 6). Nel caso in cui si eseguono n operazioni, abbiamo che il costo nel caso pessimo di una operazione è $O(n)$; questo comporta che il limite superiore del tempo di esecuzione totale di n operazioni è $O(n^2)$.

Questo limite non è stretto perché, durante l'esecuzione di n operazioni di TABLE-INSERT, non è molto frequente dover eseguire una espansione (con il relativo costo). Infatti, la i-esima operazione origina una espansione solo se $i - 1$ è una potenza esatta di 2. Il costo ammortizzato di una operazione è quindi $O(1)$, come si può dimostrare con il metodo degli aggregati. Il costo della i-esima operazione è

$$c_i = \begin{cases} i & \text{se } i - 1 \text{ è una potenza di } 2\,, \\ 1 & \text{altrimenti}\,. \end{cases}$$

Il costo totale di n operazioni TABLE-INSERT risulta

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\,, \end{aligned}$$

dato che vengono eseguite al più n operazioni di costo unitario e il costo delle rimanenti operazioni forma una serie geometrica. Il costo totale di n operazioni TABLE-INSERT è $3n$ e, quindi, il costo ammortizzato di una singola operazione è 3.

Con il metodo degli accantonamenti, riusciamo a ottenere una migliore intuizione del perché il costo ammortizzato di una operazione TABLE-INSERT debba essere 3. L'idea intuitiva è la seguente: ogni elemento nella tabella deve pagare 3 inserzioni elementari: il suo stesso inserimento nella tabella corrente, lo spostamento dovuto a una espansione della tabella e lo spostamento di un altro elemento, che era già stato spostato da un'altra tabella, quando si presenta una espansione della tabella. Per esempio, supponiamo che, immediatamente dopo una espansione, la dimensione della tabella sia m. Allora, il numero di elementi della tabella è $m/2$, e la tabella non contiene crediti. Addebitiamo tre dollari per ogni inserzione: la prima inserzione elementare costa un dollaro, un altro dollaro viene addebitato come credito dell'elemento inserito. Infine, un terzo dollaro viene addebitato come credito di uno degli $m/2$ elementi della tabella. Per completare la tabella sono necessarie altre $m/2$ operazioni di inserzione e, quindi, prima che la tabella sia piena (contenga m elementi) ogni elemento possiede un dollaro per pagare il suo spostamento ed il suo inserimento nella nuova tabella nella fase di espansione.

Anche il metodo del potenziale può essere utilizzato per analizzare una sequenza di n operazioni TABLE-INSERT; in particolar modo questo metodo sarà utilizzato nel paragrafo 18.4.2 per progettare una operazione TABLE-DELETE con un costo ammortizzato $O(1)$. Prima di tutto, definiamo una funzione potenziale Φ che vale 0 immediatamente dopo una espansione, ma che vale tanto quanto la dimensione della tabella quando la tabella è piena: la successiva espansione è pagata completamente dal potenziale. La funzione

$$\Phi(T) = 2 \cdot num[T] - size[T] \tag{18.4}$$

è una possibile definizione della funzione potenziale. Immediatamente dopo una espansione abbiamo che $num[T] = size[T]/2$ e quindi $\Phi(T) = 0$ come richiesto. Immediatamente prima di una espansione abbiamo che $num[T] = size[T]$ e quindi $\Phi[T] = num[T]$, come richiesto. Il valore iniziale del potenziale è 0 e, dato che la tabella è sempre piena almeno per metà, $num[T] \ge size[T]/2$. Questo implica che la funzione $\Phi(T)$ è sempre maggiore o uguale a zero. Pertanto, la somma dei costi ammortizzati di n operazioni TABLE-INSERT è un limite superiore della somma dei costi effettivi.

Per analizzare il costo ammortizzato della i-esima operazione TABLE-INSERT, indichiamo il numero di elementi memorizzati nella tabella dopo la i-esima operazione con num_i, la dimensione della tabella dopo la i-esima operazione con $size_i$ e il potenziale dopo la i-esima operazione con Φ_i. Inizialmente abbiamo che $num_0 = 0$, $size_0 = 0$ e $\Phi_0 = 0$.

Se la i-esima operazione TABLE-INSERT non attiva una espansione, allora $size_i = size_{i-1}$, e il costo ammortizzato dell'operazione risulta

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\ &= 3\,. \end{aligned}$$

Invece, se la i-esima operazione attiva una espansione, allora $size_i/2 = size_{i-1} = num_i - 1$, e il costo ammortizzato dell'operazione diventa

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - (2 \cdot num_i - 2)) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) \\ &= 3\,. \end{aligned}$$

La figura 18.3 mostra come i valori di num_i, $size_i$ e Φ_i cambiano al crescere di i. Si noti come il potenziale paghi il costo dell'espansione della tabella.

Figura 18.3 *L'effetto di una sequenza di n operazioni* TABLE-INSERT *sul numero num_i di elementi nella tabella, il numero $size_i$ di posizioni della tabella, e il potenziale $\Phi_i = 2 \cdot num_i - size_i$, dove ognuna di queste quantità è misurata dopo l'esecuzione della i-esima operazione. La linea sottile mostra i valori che assume num_i, quella spessa mostra $size_i$, mentre quella tratteggiata mostra Φ_i. Si noti che, immediatamente prima di una espansione, il potenziale ha un valore pari al numero degli elementi della tabella e quindi è in grado di pagare il costo del trasferimento degli elementi nella nuova tabella. Dopo l'espansione il potenziale cade a 0, ma subito dopo viene incrementato di 2 non appena viene inserito l'elemento che ha originato l'espansione.*

18.4.2 Espansione e contrazione di tabelle

Nel realizzare l'operazione TABLE-DELETE, è piuttosto semplice eliminare dalla tabella l'elemento specificato. Tuttavia, quando il fattore di carico diventa troppo piccolo, è utile ***contrarre*** la tabella in modo tale che la quantità di spazio sprecato non diventi eccessivamente elevata. L'operazione di contrazione di una tabella è l'operazione simmetrica dell'operazione di estensione di una tabella: quando il numero degli elementi della tabella diventa troppo piccolo si assegna una nuova porzione di memoria per una nuova tabella, più piccola della tabella corrente, e si copiano gli elementi della vecchia tabella nella nuova. La memoria utilizzata dalla vecchia tabella può essere restituita al sistema di gestione della memoria. Da un punto di vista ideale si vorrebbe poter preservare le due proprietà seguenti:

- il fattore di carico della tabella dinamica è limitato inferiormente da una costante,
- il costo ammortizzato di una operazione è limitato superiormente da una costante.

Assumiamo che il costo possa essere misurato in termini di inserzioni ed eliminazioni elementari.

Una strategia naturale per le operazioni di espansione e contrazione è la strategia che duplica le dimensioni della tabella allorché si deve inserire un elemento in una tabella piena, e dimezza le dimensioni quando una eliminazione ha l'effetto di far diventare la tabella piena per meno della metà. Questa semplice strategia assicura che il fattore di carico della tabella non diventi mai minore di 1/2, ma ha l'effetto spiacevole che il costo ammortizzato di una operazione è piuttosto elevato. Per esempio, si consideri lo scenario seguente. Supponiamo di eseguire n operazioni sulla tabella T e che n sia una potenza di 2. Inoltre supponiamo che le prime $n/2$ operazioni siano operazioni di inserzione, che per l'analisi fatta in precedenza hanno un costo

totale di $\Theta(n)$. Al termine di questa sequenza di inserzioni, $num[T] = size[T] = n/2$. La seconda metà di questa sequenza di operazioni è così definita:

I, D, D, I, I, D, D, I, I, ...

dove I (INSERT) indica una operazione di inserzione, mentre D (DELETE) indica una operazione di eliminazione. La prima inserzione ha l'effetto di far espandere la tabella fino alla dimensione n. Le due successive operazioni di eliminazione hanno l'effetto di fare contrarre la tabella così che la sua dimensione ritorna $n/2$. I due inserimenti successivi danno origine a una nuova espansione, e così via. Il costo di ogni espansione e di ogni contrazione è $\Theta(n)$, e ne abbiamo in totale $\Theta(n)$. Quindi, il costo totale delle n operazioni è $\Theta(n^2)$ e il costo ammortizzato di una operazione è $\Theta(n)$.

La difficoltà principale di questa strategia è chiara: dopo una espansione non si eseguono abbastanza eliminazioni per pagare il costo di una contrazione. Analogamente, dopo una contrazione non si effettuano abbastanza inserzioni per pagare il costo di una espansione.

Un miglioramento della strategia si ottiene se si permette che il fattore di carico della tabella possa diventare minore di 1/2. In questa ottica, quando si deve inserire un elemento in una tabella piena si continua a duplicare la dimensione della tabella, ma si dimezzano le dimensioni della tabella nel caso in cui una eliminazione ha l'effetto di far diventare la tabella piena per meno di un quarto, e non per meno della metà come nella strategia descritta in precedenza. Il fattore di carico della tabella è limitato inferiormente dalla costante 1/4. L'idea intuitiva è che, dopo una espansione, il fattore di carico della tabella è 1/2. In questo modo, per avere una contrazione, bisogna eliminare la metà degli elementi della tabella: infatti se il fattore di carico della tabella non raggiunge 1/4, la contrazione non avviene. Si noti che dopo una contrazione il fattore di carico della tabella è 1/2. In conclusione, prima che si possa manifestare una espansione devono essere eseguite tante operazioni di inserzione da duplicare il numero degli elementi della tabella; infatti l'espansione si manifesta solo se il fattore di carico della tabella raggiunge il valore 1.

Non riportiamo le istruzioni della procedura TABLE-DELETE dato che sono sostanzialmente identiche a quelle della procedura TABLE-INSERT. Tuttavia, per gli scopi dell'analisi conviene assumere che se il numero degli elementi della tabella diventa uguale a 0, allora lo spazio di memoria assegnato alla tabella viene liberato e restituito al sistema di gestione della memoria. In base a questa ipotesi abbiamo che $num[T] = 0$ implica che $size[T] = 0$.

A questo punto utilizziamo il metodo del potenziale per determinare il costo di una sequenza di n operazioni TABLE-DELETE e TABLE-INSERT. Prima di tutto definiamo la funzione potenziale Φ in modo tale che, subito dopo una espansione o una contrazione, il valore della funzione sia 0, e resti non negativo sia che il fattore di carico cresca fino a 1, sia che diminuisca fino a 1/4. Indichiamo con $\alpha[T]$ il fattore di carico di una tabella non vuota, dove $\alpha[T] = num[T] / size[T]$. Dato che per una tabella vuota $num[T] = size[T] = 0$ e $\alpha[T] = 1$, allora è sempre vero che $num[T] = \alpha[T]\, size[T]$, indipendentemente dal fatto che la tabella sia piena o vuota. Come funzione potenziale consideriamo la funzione così definita:

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{se } \alpha(T) \geq 1/2 , \\ size[T]/2 - num[T] & \text{se } \alpha(T) < 1/2 . \end{cases} \tag{18.5}$$

Si noti che il potenziale di una tabella vuota è 0, ed inoltre che il potenziale è sempre maggiore o uguale a zero. Pertanto, il costo ammortizzato totale di una sequenza di operazioni rispetto alla funzione Φ è un limite superiore del costo effettivo delle operazioni.

Figura 18.4 *L'effetto di una sequenza di n operazioni* TABLE-INSERT *e* TABLE-DELETE *sul numero num di elementi nella tabella, il numero $size_i$ di posizioni della tabella, e il potenziale*

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i & se\ \alpha_i \geq 1/2\ , \\ size_i/2 - num_i & se\ \alpha_i < 1/2\ , \end{cases}$$

dove ognuna di queste quantità è misurata dopo l'esecuzione della i-esima operazione. La linea sottile mostra i valori che assume num_i, quella spessa mostra $size_i$, mentre quella tratteggiata mostra Φ_i. Si noti che immediatamente prima di una espanzione il potenziale ha raggiunto un valore pari al numero di elementi nella tabella, e quindi è in grado di pagare il costo del trasferimento degli elementi nella nuova tabella. Analogamente, prima di una contrazione, il potenziale ha raggiunto un valore pari al numero di elementi nella tabella.

Prima di procedere con l'analisi puntuale, vogliamo mettere in evidenza alcune proprietà della funzione potenziale. Si noti che se il fattore di carico è 1/2, allora il potenziale è 0. Quando il fattore di carico è 1 abbiamo che $num[T] = size[T]$ e questo implica che $\Phi(T) = num[T]$: il potenziale è in grado di pagare il costo di una operazione di inserzione che causa una espansione. Quando il fattore di carico è 1/4 abbiamo che $size[T] = 4 \cdot num[T]$, e questo implica che $\Phi(T) = num[T]$: il potenziale è in grado di pagare il costo di una operazione di eliminazione che causa una contrazione della tabella. La figura 18.4 illustra il comportamento della funzione potenziale.

Prima di analizzare il costo di una sequenza di n operazioni TABLE-DELETE e TABLE-INSERT introduciamo qualche notazione ausiliaria. Indichiamo con c_i il costo effettivo della i-esima operazione, con $\hat{c}_i$ il suo costo ammortizzato rispetto alla funzione Φ, con num_i il numero degli elementi della tabella dopo la i-esima operazione, con $size_i$ la dimensione della tabella dopo la i-esima operazione, con α_i il fattore di carico dopo la i-esima operazione e infine con Φ_i il potenziale dopo la i-esima operazione. Inizialmente abbiamo che $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$ e $\Phi_0 = 0$.

Prima di tutto consideriamo il caso che la i-esima operazione sia l'operazione TABLE-INSERT. Se $\alpha_{i-1} \geq 1/2$, l'analisi ammortizzata coincide esattamente con l'analisi descritta nel paragrafo 18.4.1. Sia che avvenga o non avvenga una espansione della tabella, abbiamo che il costo ammortizzato $\hat{c}_i$ della operazione è al più 3. Se $\alpha_{i-1} < 1/2$, come effetto della operazione non abbiamo una espansione della tabella dato che l'espansione avviene solo

se $\alpha_{i-1} = 1$. Se vale che anche $\alpha_{i-1} < 1/2$ allora il costo ammortizzato della i-esima operazione è

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0\ . \end{aligned}$$

Se $\alpha_{i-1} < 1/2$ ma $\alpha_i \geq 1/2$, allora

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3 \\ &= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\ &< \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 \\ &= 3\ . \end{aligned}$$

Quindi, il costo ammortizzato di una operazione TABLE-INSERT è al più 3.

Ora possiamo considerare il caso in cui la i-esima operazione sia una operazione TABLE-DELETE e $num_i = num_{i-1} - 1$. Se $\alpha_{i-1} < 1/2$, allora si deve vedere se l'operazione ha l'effetto di attivare una contrazione. Nel caso in cui non si manifesta la contrazione, abbiamo che $size_i = size_{i-1}$ e il costo ammortizzato della operazione diventa

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2\ . \end{aligned}$$

Se $\alpha_{i-1} < 1/2$ e la i-esima operazione attiva una contrazione, allora il costo effettivo dell'operazione è $c_i = num_i + 1$, dato che si elimina un elemento e si spostano num_i elementi. Inoltre, $size_i/2 = size_{i-1}/4 = num_i + 1$ e il costo ammortizzato dell'operazione diventa

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1\ . \end{aligned}$$

Quando la i-esima operazione è un'operazione TABLE-DELETE e $\alpha_{i-1} \geq 1/2$ il costo ammortizzato è ancora limitato superiormente da una costante. Questa analisi è lasciata come esercizio (Cfr. Esercizio 18.4-3).

In conclusione, dato che il costo ammortizzato delle operazioni è limitato superiormente da una costante, il tempo effettivo per una qualunque sequenza di n operazioni su una tabella dinamica è $O(n)$.

Esercizi

18.4-1 Si mostri in modo informale che se $\alpha_{i-1} \le 1/2$ e $\alpha_i \le 1/2$ allora il costo ammortizzato di una operazione TABLE-INSERT è 0.

18.4-2 Supponiamo di voler realizzare una tabella hash dinamica a indirizzamento aperto. Per quale motivo è opportuno considerare piena una tabella il cui fattore di carico raggiunge un valore α strettamente minore di 1? Si descriva brevemente come una operazione di inserzione in una tabella hash dinamica a indirizzamento aperto venga eseguita in modo tale che il valore atteso del costo ammortizzato di una inserzione sia $O(1)$. Per quale motivo il valore atteso del costo effettivo di una inserzione non è necessariamente $O(1)$ per tutte le inserzioni?

18.4-3 Si mostri che se la i-esima operazione in una tabella dinamica è l'operazione TABLE-DELETE e $\alpha_{i-1} \ge 1/2$, allora il costo ammortizzato dell'operazione, rispetto alla funzione potenziale (18.5), è limitato superiormente da una costante.

18.4-4 Supponiamo che, invece di contrarre una tabella dimezzando le sue dimensioni quando il fattore di carico diventa più piccolo di 1/4, si effettua la contrazione della tabella moltiplicando la dimensione della tabella per 2/3 quando il fattore di carico è inferiore a 1/3. Con la funzione potenziale

$$\Phi(T) = |2 \cdot num[T] - size[T]|$$

si dimostri che il costo ammortizzato di una operazione TABLE-DELETE che utilizza la strategia precedente è limitato superiormente da una costante.

Problemi

18.1 Contatore binario dei bit rovesciati

Il Capitolo 32 discute un algoritmo molto importante conosciuto con il nome di Trasformata Veloce di Fourier (in inglese Fast Fourier Transform o FFT). Il primo passo dell'algoritmo FFT consiste in una ***permutazione dei bit rovesciati*** (bit-reversal permutation) su di un array $A[0 \,..\, n-1]$, di lunghezza $n = 2^k$, per un qualche intero positivo k. Questa permutazione scambia tutti quegli elementi i cui indici hanno rappresentazioni binarie che sono una il rovescio dell'altra.

Possiamo rappresentare un generico indice a con una sequenza di k bit $\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle$ dove $a = \sum_{i=0}^{k-1} a_i 2^i$.

Definiamo

$$rev_k(\langle a_{k-1}, a_{k-2}, \ldots, a_0 \rangle) = \langle a_0, a_1, \ldots, a_{k-1} \rangle$$

e così,

$$rev_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i .$$

Per esempio, se $n = 16$ (ovvero $k = 4$), allora $rev_k(3) = 12$, dato che la rappresentazione con cifre binarie di 4 bit del numero 3 è 0011 e il rovescio è 1100, che è la rappresentazione binaria con cifre di 4 bit di 12.

a. Data una funzione rev_k che richiede tempo $\Theta(k)$, si progetti un algoritmo che realizzi la permutazione dei bit rovesciati su un array di lunghezza $n = 2^k$ in tempo $O(nk)$.

Per migliorare le prestazioni si può utilizzare un algoritmo basato sull'analisi ammortizzata. Si introduce un "contatore dei bit rovesciati" e una procedura BIT-REVERSED-INCREMENT che, quando viene applicata a un valore a del contatore, restituisce come risultato $rev_k(rev_k(a) + 1)$. Se $k = 4$, per esempio, e il contatore dei bit rovesciati è inizializzato a 0, allora successive chiamate dell'operazione BIT-REVERSED-INCREMENT restituiscono la sequenza

0000, 1000, 0100, 1100, 0010, 1010, ... = 0, 8, 4, 12, 2, 10, ...

b. Si assuma che il sistema sia in grado di memorizzare parole di k bit, e che, in una unità di tempo, il sistema sia in grado di manipolare valori binari tramite operazioni come spostamenti di lunghezza arbitraria a sinistra o a destra (SHIFT-LEFT, SHIFT-RIGHT), AND (bit a bit) e OR (bit a bit). Si descriva una realizzazione della procedura BIT-REVERSED-INCREMENT che permetta di eseguire la permutazione dei bit rovesciati su di un array con n elementi in un tempo totale $O(n)$.

c. Supponiamo di poter eseguire uno spostamento a sinistra o a destra di una parola binaria per un solo bit alla volta in tempo unitario. In questo caso, è ancora possibile realizzare la procedura di permutazione dei bit rovesciati in un tempo totale di $O(n)$?

18.2 Ricerca binaria dinamica

La ricerca binaria su di un array ordinato viene eseguita in tempo logaritmico, ma il tempo per inserire un nuovo elemento nell'array è lineare nella dimensione dell'array. Possiamo migliorare le prestazioni dell'operazione di inserzione se consideriamo un certo numero di array ordinati.

Più precisamente, supponiamo di voler realizzare le procedure SEARCH e INSERT su un insieme di n elementi. Sia $k = \lceil \lg(n+1) \rceil$, e sia $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$ la rappresentazione binaria di n. Consideriamo k array ordinati $A_0, A_1, \ldots, A_{k-1}$, dove per $i = 0, 1, \ldots, k-1$, la lunghezza dell'array A_i è 2^i. Ogni array è pieno o vuoto, a seconda del fatto che $n_i = 1$ o $n_i = 0$, rispettivamente. Pertanto, il numero totale degli elementi presenti negli array risulta

$$\sum_{i=0}^{k-1} n_i 2^i = n.$$

Infine, anche se ogni singolo array è ordinato, non esiste alcuna relazione particolare tra gli elementi di array distinti.

a. Si descriva come realizzare l'operazione SEARCH in questa struttura di dati. Si analizzi il tempo di esecuzione nel caso pessimo.

b. Si descriva come si possa inserire un nuovo elemento in questa struttura di dati. Si analizzi il tempo di esecuzione sia nel caso pessimo che nel caso ammortizzato.

c. Si discuta la realizzazione dell'operazione DELETE.

18.3 *Alberi ammortizzati con peso bilanciato*

Si consideri un normale albero binario di ricerca dove ogni nodo x contenga un ulteriore campo, $size[x]$, che descrive il numero di chiavi memorizzate nel sottoalbero che ha il nodo x come radice. Sia α una costante che varia nell'intervallo $1/2 \le \alpha < 1$. Diciamo che un nodo x è ***α-bilanciato*** se

$$size[left[x]] \le \alpha \cdot size[x]$$

e

$$size[right[x]] \le \alpha \cdot size[x].$$

Un albero viene detto ***α-bilanciato*** se ogni nodo dell'albero è α-bilanciato. La seguente strategia di analisi ammortizzata, suggerita da G. Varghese, ha come obiettivo quello di mantenere un albero bilanciato rispetto al peso.

a. Un albero 1/2-bilanciato è il più bilanciato possibile. Dato un nodo x in un qualsiasi albero binario di ricerca, si mostri come ricostruire il sottoalbero di radice x in modo tale che il sottoalbero diventi 1/2-bilanciato. L'algoritmo proposto deve avere un tempo di esecuzione $\Theta(size[x])$ e può utilizzare $O(size[x])$ memoria ausiliaria.

b. Si mostri che una ricerca in un albero α-bilanciato con n nodi richiede tempo $O(\lg n)$ nel caso pessimo.

Per i rimanenti quesiti di questo problema assumiamo che la costante α sia strettamente maggiore di 1/2. Supponiamo che le operazioni INSERT e DELETE siano realizzate esattamente come nel caso di un albero binario di ricerca con n nodi; l'unica differenza consiste nel fatto che se, dopo aver eseguito l'operazione, alcuni nodi dell'albero non sono più α-bilanciati, allora si deve fare in modo che il sottoalbero, la cui radice è il più alto nodo non α-bilanciato, diventi 1/2-bilanciato.

Analizziamo questo metodo di ricostruzione dell'albero con l'ausilio della tecnica del potenziale. Sia x un nodo di un albero binario di ricerca T. Definiamo

$$\Delta(x) = |size[left[x]] - size[right[x]]|,$$

e definiamo il potenziale di T

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \ge 2} \Delta(x),$$

dove c è una costante sufficientemente grande che dipende da α.

c. Perché un generico albero binario di ricerca ha un potenziale non negativo e un albero 1/2-bilanciato ha un potenziale uguale a 0?

d. Supponiamo che per ricostruire un sottoalbero con m nodi sia possibile utilizzare m unità di potenziale. Quale valore deve assumere la costante c (che dipende da α) in modo tale che il tempo ammortizzato per la ricostruzione di un albero non α-bilanciato sia $O(1)$?

e. Si dimostri che l'operazione di inserzione di un nodo (o l'operazione di eliminazione di un nodo) in un albero α-bilanciato con n nodi ha costo ammortizzato $O(\lg n)$.

Note al capitolo

Il metodo degli aggregati per l'analisi ammortizzata è stato utilizzato da Aho, Hopcroft e Ullman [4]. Tarjan [189] presenta una rassegna dei metodi di accantonamento e del potenziale per l'analisi ammortizzata, e discute numerose applicazioni. Tarjan attribuisce la definizione del metodo degli accantonamenti a numerosi autori, tra gli altri M. R. Brown, R. E. Tarjan, S. Huddleston e K. Mehlhorn. Inoltre, Tarjan attribuisce la definizione del metodo del potenziale a D. D. Sleator. Il termine "analisi ammortizzata" si deve a D. D. Sleator e R. E. Tarjan.

V

Strutture di dati evolute

Introduzione

In questa parte verranno considerate nuovamente quelle strutture di dati che permettono di definire ed eseguire operazioni su insiemi dinamici. Tuttavia, in questa parte si utilizzeranno tecniche molto più sofisticate di quelle utilizzate nella Parte III. Per esempio, molti risultati presentati in due dei capitoli seguenti sono basati sull'uso delle tecniche dell'analisi ammortizzata che abbiamo introdotto nel Capitolo 18.

Il Capitolo 19 introduce la struttura di dati dei B-alberi. Questi non sono altro che alberi bilanciati di ricerca progettati per essere memorizzati su dischi magnetici. Poiché i dischi magnetici sono molto più lenti delle memorie a accesso casuale, la misura dell'efficienza di un B-albero non è data solamente dal tempo necessario per eseguire le operazioni su insiemi dinamici, ma anche da quanti accessi al disco sono stati eseguiti. Per tutte le operazioni definite sui B-alberi vale la proprietà seguente: il numero di accessi al disco aumenta con l'aumentare dell'altezza del B-albero. Pertanto tutte le operazioni definite sui B-alberi devono mantenere bassa l'altezza dell'albero.

I Capitoli 20 e 21 discutono la realizzazione di alcune operazioni su heap aggregabili quali INSERT, MINIMUM, EXTRACT-MIN e UNION. L'operazione UNION unisce, o fonde assieme, due heap. Un certo numero di altre operazioni sono disponibili su queste strutture di dati, per esempio, le operazioni DELETE e DECREASE-KEY.

Nel caso degli heap binomiali, introdotti nel Capitolo 20, tutte le operazioni descritte in precedenza possono essere eseguite in tempo $O(\lg n)$ nel caso pessimo, dove n è il numero totale di elementi dello heap in ingresso (o nei due heap di ingresso nel caso dell'operazione UNION). È proprio il fatto di eseguire l'operazione UNION in tempo $O(\lg n)$, che rende gli heap binomiali superiori agli heap binari introdotti nel Capitolo 7. Nel caso degli heap binari l'operazione UNION viene eseguita in tempo $\Theta(n)$ nel caso pessimo.

Gli heap di Fibonacci, introdotti nel Capitolo 21, almeno da un punto di vista teorico sono più efficienti degli heap binomiali. Per valutare l'efficienza degli heap di Fibonacci faremo uso di limiti di tempo ammortizzati. Le operazioni INSERT, MINIMUM e UNION sugli heap di Fibonacci richiedono esattamente un tempo, sia ammortizzato che effettivo, di $O(1)$. Le operazioni EXTRACT-MIN e DELETE richiedono tempo ammortizzato $O(\lg n)$. Il vantaggio maggiormente significativo degli heap di Fibonacci è che l'operazione DECREASE-KEY richiede solo un tempo ammortizzato di $O(1)$. Il basso valore del tempo ammortizzato dell'operazione DECREASE-KEY è il motivo per cui gli heap di Fibonacci sono alla base di alcuni dei più veloci, in senso asintotico, algoritmi per problemi su grafi definiti fino a ora.

Infine il Capitolo 22 introduce alcune strutture di dati per insiemi disgiunti. Si consideri un universo con n elementi raggruppati in un certo numero di insiemi dinamici. Inizialmente ogni elemento definisce un singolo insieme. L'operazione UNION unisce due insiemi mentre l'interrogazione FIND-SET identifica l'insieme che contiene un determinato elemento. Se rappresentiamo ogni insieme come un semplice albero radicato, allora otteniamo operazioni particolarmente efficienti: una sequenza di m operazioni richiede un tempo $O(m\,\alpha(m, n))$, dove $\alpha(m, n)$ è una funzione che cresce molto lentamente, tanto lentamente che se n è una stima del numero di atomi presenti nell'universo conosciuto, allora $\alpha(m, n)$ è al massimo 4. L'analisi ammortizzata che dimostra questo limite di tempo è tanto complicata quanto la struttura di dati è semplice. Nel Capitolo 22 si dimostra che vale un interessante e sostanzialmente più semplice limite del tempo di esecuzione.

Le strutture di dati introdotte in questa parte non sono le sole strutture di dati "evolute". Tra le numerose proposte di strutture di dati evolute ricordiamo le seguenti:

- La struttura di dati introdotta da van Emde Boas [194] che fornisce le operazioni MINIMUM, MAXIMUM, INSERT, DELETE, SEARCH, EXTRACT-MIN, EXTRACT-MAX, PREDECESSOR e SUCCESSOR. Queste operazioni, nel caso in cui l'universo delle chiavi è l'insieme $\{1, 2, \ldots, n\}$, hanno tempo di esecuzione $O(\lg(\lg n))$ nel caso pessimo.

- Gli ***alberi dinamici***, introdotti da Sleator e Tarjan [177] e discussi da Tarjan [188], hanno la caratteristica di considerare una foresta di alberi con radici disgiunte. Ogni arco di un singolo albero è caratterizzato da un costo a valore reale. Gli alberi dinamici sono in grado di rispondere a interrogazioni di ricerca del padre, della radice, dei costi degli archi, dell'arco di costo minimo in un cammino che dalla radice raggiunge un determinato nodo. Questi alberi possono essere modificati da operazioni che eliminano degli archi, modificano il costo degli archi in un cammino che va dalla radice fino a un determinato nodo, collegano una radice a un albero diverso, trasformano un nodo qualunque in una radice. Una semplice realizzazione degli alberi dinamici fornisce un limite di tempo ammortizzato $O(\lg n)$ per tutte le operazioni elencate in precedenza: una realizzazione più sofisticata porta ad avere un limite di tempo di $O(\lg n)$ nel caso pessimo.

- Gli ***alberi splay***, sviluppati da Sleator e Tarjan [178] e discussi da Tarjan [188], costituiscono una forma di albero binario di ricerca dove le tipiche operazioni di ricerca su alberi sono eseguite in tempo ammortizzato $O(\lg n)$. Gli alberi splay possono essere visti come un caso particolarmente semplice degli alberi dinamici.

- Le strutture di dati ***persistenti*** permettono di fare interrogazioni, e qualche volta anche modifiche, a versioni precedenti di una struttura di dati. Driscoll, Sarnak, Sleator e Tarjan [59] presentano un certo numero di tecniche che permettono di costruire strutture persistenti di dati collegati con piccoli costi sia in spazio che in tempo. Il Problema 14.1 discute un semplice esempio di insieme persistente dinamico.

B-Alberi

19

I B-alberi sono alberi bilanciati di ricerca progettati per operazioni su dischi magnetici o altri dispositivi di memoria secondaria a accesso diretto. I B-alberi hanno caratteristiche comuni con gli alberi rosso-neri (alberi RB introdotti nel Capitolo 14), ma, rispetto a questi ultimi, sono particolarmente efficienti nelle operazioni di Input/Output su dischi magnetici.

La principale differenza tra i B-alberi e gli alberi RB è che i nodi dei B-alberi possono avere molti figli: da poche decine a diverse centinaia. In altri termini, il "grado" o "fattore di ramificazione" di un B-albero può essere particolarmente elevato: in genere è determinato dalle caratteristiche della unità disco. Abbiamo detto che i B-alberi presentano delle caratteristiche che li rendono simili agli alberi RB: infatti ogni B-albero con n nodi ha altezza $O(\lg n)$, anche se l'altezza di un B-albero può essere molto minore dell'altezza di un albero RB, poiché il suo grado di ramificazione può essere piuttosto elevato. Inoltre, i B-alberi possono essere utilizzati per realizzare un gran numero di operazioni su insiemi dinamici in tempo $O(\lg n)$.

I B-alberi sono la naturale generalizzazione degli alberi binari di ricerca. La figura 19.1 mostra un (semplice) B-albero. Se un nodo x di un B-albero contiene un numero $n[x]$ di chiavi, allora x ha $n[x] + 1$ figli. Le chiavi di un nodo x sono sostanzialmente dei punti di divisione: dividono l'intervallo di chiavi gestito dal nodo x in $n[x] + 1$ (sotto)intervalli, dove ciascun sottointervallo è gestito da un figlio di x. Durante un'operazione di ricerca di una chiave in un B-albero, ogni nodo x presenta $n[x] + 1$ scelte alternative: la scelta dell'alternativa dipende dal confronto della chiave cercata con le $n[x]$ chiavi memorizzate nel nodo.

I B-alberi vengono definiti formalmente nel paragrafo 19.1; si dimostra che l'altezza di un B-albero cresce in modo logaritmico con il numero dei nodi dell'albero. Nel paragrafo 19.2 si descrivono le operazioni di ricerca di una chiave e di inserimento di una chiave in un B-albero; infine nel paragrafo 19.3 si discute il problema della eliminazione. Tuttavia, prima di

Figura 19.1 *Un B-albero le cui chiavi sono le consonanti dell'alfabeto inglese. Un nodo interno x contiene $n[x]$ chiavi e ha $n[x] + 1$ figli. Tutte le foglie dell'albero sono alla stessa profondità. I nodi leggermente più chiari sono quelli che vengono esaminati in una ricerca della lettera R.*

Figura 19.2 *La struttura di un disco magnetico.*

procedere nello studio dei B-alberi, dobbiamo rispondere alla domanda seguente: per quale motivo le strutture di dati progettate per essere memorizzate su dischi magnetici sono valutate in modo diverso dalle strutture di dati progettate per essere memorizzate in una memoria principale ad accesso casuale?

Strutture di dati su memorie secondarie

Le tecnologie disponibili per fornire a un sistema[1] una certa capacità di memoria sono molto diverse. La ***memoria primaria*** (o ***memoria principale***) di un sistema generalmente consiste di un certo numero di chip di memoria, ciascuno dei quali può contenere dati per almeno 1 milione di bit. Questa tecnologia è più costosa (a parità di bit memorizzati) della tecnologia delle memorie magnetiche, come nastri e dischi. Un qualunque sistema è sempre equipaggiato con una ***memoria secondaria*** basata su dischi magnetici: generalmente la capacità della memoria secondaria supera di alcuni ordini di grandezza la capacità della memoria principale.

La figura 19.2 mostra la tipica struttura di un disco. La superficie del disco è ricoperta da materiale magnetico. La testina di lettura/scrittura è in grado di leggere o scrivere dati sulla superficie del disco mentre il disco è in movimento. Il braccio di lettura/scrittura è in grado di posizionare la testina in un certo numero di posizioni lungo il raggio che porta al centro del disco. Quella porzione di superficie che passa sotto la testina, in una data posizione, viene detta ***traccia***. L'informazione memorizzata su una singola traccia viene comunemente divisa in un numero fisso di ***pagine*** della stessa dimensione: una pagina può contenere, per esempio, 2048 byte. La pagina definisce la quantità unitaria di informazione che viene letta e scritta: in altri termini, le scritture e le letture da disco sono formate da pagine intere. Il tempo necessario per posizionare la testina di lettura e scrittura e per aspettare che una determinata pagina di informazione passi sotto la testina di lettura viene detto ***tempo di accesso***. Di solito il tempo di accesso è piuttosto elevato (per esempio 20 millisecondi), mentre il tempo necessario per leggere e scrivere una pagina (dopo l'accesso) è piccolo. La tecnologia delle memorie magnetiche è poco costosa ma il tempo necessario per accedere ai dati è relativamente lungo. Questo è il prezzo che si deve pagare per avere memorie magnetiche capienti. Infatti, muovere elettroni è molto più facile (ed economico) che muovere grandi (o anche piccoli) oggetti meccanici: i dispositivi di memoria interamente elettronici, come i chip di memoria, hanno un tempo di accesso molto minore dei dispositivi di memoria che devono

[1] N.d.T. Nel seguito la parola *sistema* verrà utilizzata per indicare un elaboratore elettronico nel suo complesso.

fare muovere alcune componenti meccaniche, come i dischi. Tuttavia, dopo che la testina si è posizionata sulla traccia corretta, la lettura o la scrittura di un disco magnetico è completamente elettronica (se si esclude il movimento rotatorio del disco), e, pertanto, grosse quantità di dati possono essere lette e scritte velocemente.

Spesso, il tempo necessario per accedere e leggere una pagina di informazione memorizzata in un disco magnetico è superiore al tempo necessario all'elaboratore per esaminare tutta l'informazione letta. Per questo motivo, in questo capitolo considereremo separatamente le due componenti principali del tempo di esecuzione:

- il numero di accessi al disco, e
- il tempo (di calcolo) dell'unità centrale di elaborazione, cioè il tempo di CPU.

Il numero di accessi al disco viene misurato in termini del numero delle pagine di informazione che devono essere lette o scritte nel disco. Si noti che il tempo di accesso al disco non è un valore costante: dipende dalla distanza tra la traccia corrente e la traccia desiderata, e anche dal tempo di latenza legato alla rotazione del disco. Nonostante questa osservazione, il numero di pagine lette e scritte può essere visto come una buona approssimazione del tempo totale necessario per accedere all'informazione memorizzata nel disco.

Nelle tipiche applicazioni dei B-alberi, la quantità di dati che viene elaborata è così elevata che la memoria principale non è sufficiente. Gli algoritmi definiti sui B-alberi copiano le pagine selezionate dal disco nella memoria principale, successivamente scrivono nel disco le pagine che sono state modificate. Gli algoritmi per i B-alberi richiedono che la memoria principale contenga, in ogni istante, solo un numero costante di pagine, pertanto la dimensione della memoria principale non vincola assolutamente la dimensione dei B-alberi su cui si deve operare.

Possiamo descrivere le operazioni del disco nel modo seguente. Sia x un puntatore a un oggetto. Se l'oggetto è già presente nella memoria principale, allora possiamo accedere ai campi dell'oggetto nel modo usuale, per esempio con il campo $key[x]$. Invece, se l'oggetto indicato da x risiede sul disco, allora prima di poter accedere ai campi dell'oggetto si deve portare l'oggetto in memoria. Per fare questo si chiama la procedura DISK-READ(x). (Si assume che se l'oggetto risiede nella memoria principale, allora l'operazione DISK-READ(x) non richiede alcun accesso al disco: si comporta come una operazione "nulla".) L'operazione DISK-WRITE(x) viene utilizzata per salvare le modifiche che sono state fatte ai campi dell'oggetto x. Pertanto, la modalità tipica per operare su un oggetto risulta:

```
...
x ← un puntatore a un qualche oggetto
DISK-READ(x)
operazioni che leggono e/o modificano i campi di x
DISK-WRITE(x)          ▷ omessa se non si modificano campi di x
altre operazioni che accedono ma non modificano campi di x
...
```

In ogni istante, il sistema è in grado di mantenere nella memoria principale solo un numero limitato di pagine. In questo capitolo, assumiamo che il sistema scarichi dalla memoria tutte

1 *nodo*.
1000 *chiavi*

1001 *nodi*.
1 001 000 *chiavi*

1 002 001 *nodi*.
1 002 001 000 *chiavi*

Figura 19.3 *Un B-albero di altezza 2 che contiene più di un miliardo di chiavi. Ogni nodo interno e ogni foglia può contenere 1000 chiavi. Ci sono 1001 nodi a profondità 1, e più di un milione di foglie a profondità 2. Nella figura, all'interno di ogni nodo, è mostrato il campo n[x], cioè il numero di chiavi del nodo.*

le pagine che non sono più in uso; gli algoritmi per i B-alberi che presenteremo ignoreranno totalmente questo aspetto.

Nella maggior parte dei sistemi, il tempo di esecuzione di un algoritmo per un B-albero è determinato dal numero di operazioni di DISK-READ e DISK-WRITE che sono state eseguite dall'algoritmo. È pratica comune fare in modo che queste operazioni siano utilizzate in modo intensivo per leggere e scrivere la maggiore quantità possibile di informazioni. Per questo motivo, un nodo di un B-albero di solito è grande tanto quanto una pagina di disco: quindi, il numero dei figli di un nodo di un B-albero è limitato dalla dimensione della pagina del disco.

Per un B-albero di grosse dimensioni memorizzato su un disco, il grado di ramificazione può variare tra 50 e 2000, e dipende dalla dimensione relativa della chiave e della pagina del disco. Un grado di ramificazione elevato riduce sensibilmente sia l'altezza dell'albero, che il numero di accessi necessari a trovare una chiave qualsiasi. La figura 19.3 mostra un B-albero con un grado di ramificazione di 1001 e una altezza uguale a 2; questo B-albero è in grado di memorizzare più di un miliardo di chiavi; tuttavia, dato che il nodo radice deve essere sempre tenuto nella memoria principale, al più sono necessari *due* soli accessi al disco per trovare una qualsiasi chiave!

19.1 Definizione dei B-alberi

Per mantenere le cose semplici, assumiamo, come abbiamo fatto per gli alberi binari di ricerca e gli alberi RB, che "l'informazione satellite" associata a una chiave sia memorizzata nello stesso nodo della chiave. In pratica, assieme a ogni chiave si potrebbe memorizzare un puntatore a quella pagina del disco che contiene l'informazione satellite della chiave. Tutte le procedure descritte in questo capitolo assumono implicitamente che l'informazione satellite associata a una chiave, o il puntatore all'informazione satellite, sia sempre legata alla chiave anche se la chiave viene spostata in un altro nodo dell'albero. In qualche realizzazione dei B-alberi l'informazione satellite è memorizzata nelle foglie, mentre nei nodi interni vengono memorizzate solo le chiavi ed i puntatori ai figli: in questo modo si massimizza il grado di ramificazione dei nodi interni.

Un ***B-albero*** è un albero radicato (*root*[*T*] è la radice dell'albero) che soddisfa le proprietà seguenti.

1. Ogni nodo x è catterizzato dai seguenti campi o attributi:
   a. $n[x]$, il numero delle chiavi memorizzate nel nodo x
   b. le $n[x]$ chiavi sono memorizzate in ordine non decrescente: $key_1[x] \le key_2[x] \le \ldots \le key_{n[x]}[x]$
   c. $leaf[x]$, un valore boleano che è uguale a TRUE se il nodo x è una foglia, è uguale a FALSE se il nodo x è un nodo interno.
2. Un nodo interno x contiene $n[x] + 1$ puntatori, $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$, ai suoi figli. Le foglie non hanno figli, quindi i campi c_i delle foglie sono sempre indefiniti.
3. I campi $key_i[x]$ definiscono gli intervalli delle chiavi memorizzate in ciascun sottoalbero: se k_i è una qualunque chiave memorizzata nel sottoalbero di radice $c_i[x]$, allora vale che $k_1 \le key_1[x] \le k_2 \le key_2[x] \le \ldots \le key_{n[x]}[x] \le k_{n[x]+1}$.
4. Tutte le foglie sono alla stessa profondità, che coincide con l'altezza dell'albero.
5. Il numero di chiavi che può essere memorizzato in un nodo è limitato sia inferiormente che superiormente. Questi limiti possono essere espressi in termini di un intero $t \ge 2$ chiamato il ***grado minimo*** del B-albero:
   a. Ogni nodo, con la sola eccezione della radice, deve contenere almeno $t - 1$ chiavi. Ogni nodo interno, con la sola eccezione della radice, deve avere almeno t figli. Se l'albero non è vuoto, la radice deve contenere almeno una chiave.
   b. Ogni nodo può contenere al massimo $2t - 1$ chiavi. Quindi un nodo interno può avere al massimo $2t$ figli. Diciamo che un nodo è ***pieno*** se contiene esattamente $2t - 1$ chiavi.

I più semplici B-alberi sono quelli in cui $t = 2$: ogni nodo interno ha 2 oppure 3 oppure 4 figli. Questi B-alberi sono noti con il nome di ***alberi 2-3-4***. In pratica, il valore di t è di solito alquanto più elevato.

L'altezza dei B-alberi

Per la maggior parte delle operazioni definite sui B-alberi, il numero di accessi al disco è proporzionale all'altezza del B-albero. Vediamo ora qual è l'altezza di un B-albero contenente n chiavi nel caso pessimo. Vale il seguente teorema.

Teorema 19.1

Se $n \ge 1$, allora per ogni B-albero T di altezza h e di grado minimo $t \ge 2$ vale

$$h \le \log_t \frac{n+1}{2} .$$

Dimostrazione. Se un B-albero ha altezza h, allora il numero dei suoi nodi risulta minimo nel caso in cui la radice contenga una sola chiave e tutti gli altri nodi contengano $t - 1$ chiavi. In questo caso ci sono 2 nodi a profondità 1, $2t$ nodi a profondità 2, $2t^2$ nodi a profondità 3 e così via fino alla profondità h dove abbiamo $2t^{h-1}$ nodi.

Figura 19.4 *Un B-albero di altezza 3 che contiene il minor numero possibile di chiavi. All'interno di ogni nodo x è mostrato n[x].*

La figura 19.4 fa vedere un B-albero con queste caratteristiche nell'ipotesi che $h = 3$. Pertanto, il numero n delle chiavi deve soddisfare la disuguaglianza

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 , \end{aligned}$$

che implica l'enunciato del teorema. ■

Possiamo subito mettere a confronto i B-alberi con gli alberi RB. Sebbene l'altezza cresca come $O(\lg n)$ per entrambe le classi di alberi (ricordiamo che t è una costante), nel caso dei B-alberi la base del logaritmo può essere molto più grande. Quindi, le operazioni definite sui B-alberi esaminano un numero di nodi inferiore, di un fattore dell'ordine di $\lg t$, al numero di nodi esaminato dalle operazioni definite sugli alberi RB. Dato che per esaminare un nodo di un B-albero occorre effettuare un accesso al disco, il numero di accessi al disco viene sostanzialmente ridotto.

Esercizi

19.1-1 Perché non si permette un grado minimo $t = 1$?

19.1-2 Per quali valori di t l'albero di figura 19.1 è un B-albero lecito?

19.1-3 Si mostrino tutti i B-alberi leciti con grado minimo uguale a 2 che rappresentano l'insieme di chiavi {1, 2, 3, 4, 5}.

19.1-4 Si derivi un limite superiore stretto del numero delle chiavi che possono essere memorizzate in un B-albero di altezza h in funzione del grado minimo t.

19.1-5 Si descriva quale tipo di struttura di dati si otterrebbe se ogni nodo nero di un albero RB incorporasse i suoi figli rossi, e avesse come figli i figli di questi ultimi, più l'eventuale proprio figlio nero.

19.2 Operazioni di base sui B-alberi

In questo paragrafo descriveremo le operazioni B-TREE-SEARCH, B-TREE-CREATE e B-TREE-INSERT. Nella definizione delle procedure adotteremo le convenzioni seguenti:

- La radice del B-albero è sempre contenuta nella memoria principale, pertanto non è mai necessario eseguire una operazione DISK-READ per leggere la radice. Invece, l'operazione DISK-WRITE deve essere eseguita ogni volta che si modifica il nodo radice.
- Tutti i nodi che sono passati come parametro alle procedure sono già presenti in memoria principale: sono state già eseguite le opportune chiamate dell'operazione DISK-READ.

Le procedure che presenteremo sono tutti algoritmi a "singola passata": algoritmi che visitano l'albero a partire dalla radice e non devono mai tornare sui propri passi.

Ricerca su B-alberi

L'operazione di ricerca fornita dai B-alberi assomiglia molto all'operazione di ricerca su alberi bilanciati di ricerca, la differenza principale consiste nel fatto che le possibili alternative, che possono essere seguite a ogni nodo, non sono due ("decisioni a due vie") ma coincidono con il numero dei figli del nodo in esame. Più precisamente, a ogni nodo interno x si presentano $n[x] + 1$ scelte alternative.

La procedura B-TREE-SEARCH è una generalizzazione immediata della procedura TREE-SEARCH definita per gli alberi binari di ricerca. I parametri di ingresso della procedura B-TREE-SEARCH sono un puntatore alla radice x di un sottoalbero ed una chiave k che deve essere ricercata in quel sottoalbero. La chiamata più esterna della procedura è quindi del tipo B-TREE-SEARCH($root[T]$, k). Se la chiave k è presente nel B-albero, la procedura B-TREE-SEARCH restituisce la coppia ordinata (y, i) che consiste di un nodo y e di un indice i tale che $key_i[y] = k$. Nel caso contrario la procedura restituisce il valore NIL.

```
B-TREE-SEARCH(x, k)
i ← 1
while i ≤ n[x] e k > key_i[x]
        do i ← i + 1
if i ≤ n[x] e k = key_i[x]
        then return (x, i)
if leaf[x]
        then return NIL
        else DISK-READ(c_i[x])
                return B-TREE-SEARCH(c_i[x], k)
```

Una ricerca lineare (linee 1-3) ha il compito di trovare il più piccolo indice i tale che $k \leq key_i[x]$, se un tale indice non esiste allora i assume il valore $n[x] + 1$. Le istruzioni delle linee 4-5 hanno il compito di controllare se è stata trovata la chiave richiesta: se i controlli hanno successo si restituisce la chiave cercata ed il nodo a cui appartiene. In caso contrario (linee 6-9) o si perviene a una foglia, e quindi si constata l'insuccesso della ricerca, oppure la ricerca prosegue con una chiamata ricorsiva della procedura su un opportuno sottoalbero del nodo in esame. Si noti che prima della chiamata ricorsiva è necessario eseguire l'operazione DISK-READ per portare in memoria la radice del sottoalbero prescelto.

La figura 19.1 mostra le operazioni della procedura B-TREE-SEARCH: i nodi leggermente più chiari sono quelli che vengono esaminati in una ricerca della lettera R.

Come per la procedura TREE-SEARCH degli alberi binari di ricerca, un B-albero viene percorso lungo un cammino che porta dalla radice alle foglie. Si vede facilmente che il numero di accessi al disco della procedura B-TREE-SEARCH è dato da $\Theta(h) = \Theta(\log_t n)$, dove h è l'altezza e n è il numero di chiavi del B-albero. Poiché $n[x] < 2t$ abbiamo che il tempo impiegato dal ciclo **while** (linee 2-3) per esaminare un nodo qualsiasi è $O(t)$, pertanto il tempo totale di CPU è dato da $O(th) = O(t \log_t n)$.

Creazione di un B-albero vuoto

La costruzione di un B-albero è il risultato di una determinata sequenza di operazioni: prima si deve chiamare la procedura B-TREE-CREATE che crea un nodo radice vuoto; successivamente, per aggiungere le nuove chiavi, si deve chiamare la procedura B-TREE-INSERT. Entrambe le procedure utilizzano la procedura ausiliaria ALLOCATE-NODE; questa procedura crea un nuovo nodo e gli assegna una opportuna pagina del disco in tempo $O(1)$. Si può assumere, senza perdere in generalità, che la procedura ALLOCATE-NODE non chiami al suo interno la procedura DISK-READ: la pagina del disco associata al nuovo nodo non contiene ancora nessuna informazione significativa.

```
B-TREE-CREATE
1  x ← ALLOCATE-NODE()
2  leaf[x] ← TRUE
3  n[x] ← 0
4  DISK-WRITE(x)
5  root[T] ← x
```

La procedura B-TREE-CREATE richiede $O(1)$ operazioni su disco ed un tempo $O(1)$ di CPU.

Divisione di un nodo in un B-albero

L'operazione di inserzione di una chiave in un B-albero è decisamente più complicata dell'operazione che inserisce una chiave in un albero binario di ricerca. L'operazione cruciale dell'inserzione è l'operazione che ***divide*** un nodo pieno y (un nodo con $2t - 1$ chiavi) all'altezza della sua ***chiave mediana*** $key_t[y]$. Il risultato di questa operazione è che il nodo y è diviso in due nodi con $t - 1$ chiavi ciascuno.

Figura 19.5 *La divisione di un nodo con $t = 4$. Il nodo y è diviso in due nodi, y e z, e la chiave mediana S di y viene spostata nel padre di y.*

La chiave mediana viene spostata nel padre del nodo y – che naturalmente non deve essere pieno – in modo tale che si possa identificare il punto di divisione dei due nuovi alberi. Se il nodo y è il nodo radice, allora l'altezza dell'albero aumenta di una unità: l'operazione di divisione di un nodo è il meccanismo di crescita degli alberi.

La procedura B-TREE-SPLIT-CHILD ha come parametri un nodo interno x *non pieno* (si assume che questo nodo sia già presente nella memoria principale), un indice i, e un nodo y tale che $y = c_i[x]$ è un figlio *pieno* di x. La procedura divide questo nodo figlio in due nodi e poi modifica opportunamente la struttura del nodo x per tenere traccia della presenza di un nuovo figlio.

La figura 19.5 mostra un esempio di applicazione di questa procedura. Il nodo pieno y è diviso lungo la sua chiave mediana S. la chiave mediana è spostata nel nodo x (il padre di y). Quelle chiavi di x che sono più grandi della chiave mediana sono inserite nel nuovo nodo z e il nodo z diventa un nuovo figlio di x.

```
B-TREE-SPLIT-CHILD(x, i, y)
1   z ← ALLOCATE-NODE()
2   leaf[z] ← leaf[y]
2   n[z] ← t − 1
4   for j ← 1 to t − 1
5       do key_j[z] ← key_{j+t}[y]
6   if not leaf[y]
7       then for j ← 1 to t
8           do c_j[z] ← c_{j+t}[y]
9   n[y] ← t − 1
10  for j ← n[x] + 1 downto i + 1
11      do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14      do key_{j+1}[x] ← key_j[x]
15  key_i[x] ← key_t[y]
16  n[x] ← n[x] + 1
```

17 DISK-WRITE(y)
18 DISK-WRITE(z)
19 DISK-WRITE(x)

La procedura B-TREE-SPLIT-CHILD si comporta esattamente come una procedura "taglia e incolla". Supponiamo che y sia il figlio i-esimo di x e che sia pieno: il nodo y deve essere diviso. In origine il nodo y ha esattamente $2t - 1$ figli, dopo la divisione il numero dei suoi figli è ridotto a $t - 1$. Il nodo z "adotta" i $t - 1$ figli di y più grandi, ed inoltre diventa un nuovo figlio di x, inserendosi subito dopo y nella tabella dei figli di x. La chiave mediana di y viene spostata e diventa quella chiave che separa y da z in x.

Le istruzioni delle linee 1-8 hanno il compito di creare il nodo z e di assegnargli le $t - 1$ chiavi di y più grandi, assieme ai figli corrispondenti. Infine, le istruzioni delle linee 10-16 hanno diversi compiti. Prima si deve inserire il nodo z come figlio di x, successivamente si deve spostare la chiave mediana di y nel nodo x in modo tale da separare y da z; infine si deve modificare il contatore delle chiavi del nodo x. Le istruzioni delle linee 17-19, infine, hanno il compito di riportare sul disco tutte le modifiche effettuate. Il tempo di CPU richiesto dalla B-TREE-SPLIT-CHILD è $\Theta(t)$, per la presenza dei due cicli nelle linee 4-5 e 7-8. (Gli altri cicli eseguono anch'essi $\Theta(t)$ iterazioni.)

Inserimento di una chiave in un B-albero

L'operazione che inserisce una chiave k in un B-albero di altezza h viene fatta con una singola visita dell'albero che richiede $O(h)$ accessi al disco. Invece il tempo di CPU richiesto è $O(th) = O(t \log_t n)$. La procedura B-TREE-INSERT utilizza la procedura B-TREE-SPLIT-CHILD per garantire che durante la ricorrenza non si inserisca mai in un nodo già pieno.

```
B-TREE-INSERT(T, k)
 1  r ← root[T]
 2  if n[r] = 2t − 1
 3      then  s ← ALLOCATE-NODE()
 4            root[T] ← s
 5            leaf[s] ← FALSE
 6            n[s] ← 0
 7            c1[s] ← r
 8            B-TREE-SPLIT-CHILD(s, 1, r)
 9            B-TREE-INSERT-NONFULL(s, k)
10      else  B-TREE-INSERT-NONFULL(r, k)
```

Le istruzioni delle linee 3-9 hanno il compito di gestire il caso in cui il nodo radice r sia pieno: in questo caso la radice è divisa in due, e il nuovo nodo s (con due figli) diventa la nuova radice. Abbiamo già ricordato che l'operazione che divide la radice è l'unica operazione che fa aumentare di una unità l'altezza del B-albero. La figura 19.6 descrive questa situazione. Si noti che, a differenza degli alberi binari di ricerca, un B-albero cresce in altezza nella radice invece che nelle foglie. La procedura B-TREE-INSERT si conclude con la chiamata della B-

Figura 19.6 *La divisione della radice con $t = 4$. Il nodo radice r viene diviso in due e quindi si crea un nuovo nodo radice. La nuova radice contiene la chiave mediana di r e ha come figli le due metà di r. Il B-albero cresce in altezza di 1 come risultato della divisione della radice.*

TREE-INSERT-NONFULL che ha il compito di inserire la chiave k nell'albero la cui radice non è piena. La procedura B-TREE-INSERT-NONFULL effettua una visita ricorsiva dell'albero; la procedura è organizzata in modo tale che prima di ogni chiamata ricorsiva si è sicuri che il nodo che si deve visitare non sia pieno. Se si incontra un nodo pieno si deve chiamare la procedura B-TREE-SPLIT-CHILD.

La procedura ausiliaria B-TREE-INSERT-NONFULL inserisce la chiave k in un nodo x, che al momento della chiamata non deve essere pieno. La struttura delle procedure B-TREE-INSERT e B-TREE-INSERT-NONFULL ci assicura che quest'ultima proprietà è sempre soddisfatta.

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3      then  while i ≥ 1 e k < key_i[x]
 4                do key_{i+1}[x] ← key_i[x]
 5                   i ← i − 1
 6            key_{i+1}[x] ← k
 7            n[x] ← n[x] + 1
 8            DISK-WRITE(x)
 9      else  while i ≥ 1 e k < key_i[x]
10                do i ← i − 1
11            i ← i + 1
12            DISK-READ(c_i[x])
12            if n[c_i[x]] = 2t − 1
14                then  B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                      if k > key_i[x]
16                         then i ← i + 1
17            B-TREE-INSERT-NONFULL(c_i[x], k)
```

La procedura B-TREE-INSERT-NONFULL opera nel modo seguente. Le istruzioni delle linee 3-8 affrontano il problema di inserire la chiave k in un nodo x nel caso in cui x sia una foglia. Se x non è una foglia, allora si deve inserire la chiave k nell'opportuna foglia del sottoalbero

Figura 19.7 *L'inserzione di chiavi in un B-albero. Il grado minimo t del B-albero in figura è 3 pertanto un nodo può contenere al massimo 5 chiavi. I nodi che sono modificati dalla procedura sono leggermente più chiari degli altri nodi.* **(a)** *La struttura iniziale dell'albero dell'esempio.* **(b)** *L'albero che si ottiene inserendo la chiave B nell'albero iniziale: questa è una semplice inserzione in una foglia.* **(c)** *L'albero che si ottiene inserendo la chiave Q nell'albero ottenuto al passo precedente. Il nodo RSTUV viene diviso nei due nodi RS e UV, la chiave T viene spostata nella radice e la chiave Q viene inserita nella metà di sinistra (il nodo RS).* **(d)** *L'albero che si ottiene inserendo la chiave L nell'albero ottenuto al passo precedente. La radice dell'albero viene divisa (la radice è piena) e il B-albero cresce in altezza di una unità. Successivamente la chiave L è inserita nella foglia che contiene le chiavi JK.* **(e)** *L'albero che si ottiene inserendo la chiave F nell'albero ottenuto al passo precedente. Prima di inserire la chiave F si deve dividere il nodo ABCDE, la chiave F è inserita nella metà di destra (il nodo DE).*

la cui radice è il nodo x. In questo caso, le istruzioni delle linee 9-11 hanno il compito di determinare il figlio del nodo x su cui si deve procedere con la ricorrenza. L'istruzione della linea 13 ha il compito di controllare se la ricorrenza deve prendere in esame un nodo pieno; se questo è il caso, si deve dividere quel nodo (la linea 14 contiene la chiamata alla B-Tree-Split-Child) e successivamente si determina quale dei due figli deve essere visitato (linee 15-16). Si noti che non c'è alcun bisogno di chiamare la procedura Disk-Read($c_i[x]$) dopo l'incremento alla variabile i (linea 16): infatti la ricorrenza prenderà in esame, in questo caso, il nodo creato dalla procedura B-Tree-Split-Child. Come risultato delle istruzioni delle linee 13-16 abbiamo

che la procedura non è mai chiamata ricorsivamente su un nodo pieno. Infine, la linea 17 contiene la chiamata ricorsiva della procedura. La figura 19.7 descrive le possibili situazioni che si devono affrontare durante l'operazione di inserimento di una chiave in un B-albero.

Nel caso di un B-albero di altezza h la procedura B-Tree-Insert effettua $O(h)$ accessi al disco; infatti le chiamate della procedura B-Tree-Insert-Nonfull eseguono solamente $O(1)$ operazioni Disk-Read e Disk-Write. Il tempo totale di CPU è $O(th) = O(t \log_t n)$. È facile vedere che la procedura B-Tree-Insert-Nonfull ha una struttura di ricorrenza in coda, pertanto può essere realizzata con un ciclo **while**; in questo modo si dimostra anche che il numero di pagine che devono risiedere simultaneamente nella memoria principale è $O(1)$.

Esercizi

19.2-1 Si mostrino i risultati dell'inserzione della sequenza di chiavi

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

in un B-albero vuoto. Si mostri solamente la configurazione dell'albero prima della divisione di qualche nodo. Inoltre si mostri la configurazione finale.

19.2-2 In quali casi potrebbe accadere di eseguire delle chiamate ridondanti alle procedure Disk-Read e Disk-Write durante una chiamata della procedura B-Tree-Insert? Si motivi la risposta. (Una chiamata di Disk-Read viene detta ridondante se quella pagina è già presente nella memoria principale. Una chiamata di Disk-Write viene detta ridondante se si scrive sul disco una pagina identica a quella che è memorizzata nel disco.)

19.2-3 Si spieghi come trovare la chiave più piccola presente in un B-albero, e come trovare il predecessore di una determinata chiave memorizzata in un B-albero.

* *19.2-4* Supponiamo di inserire le chiavi $\{1, 2, \ldots, n\}$ in un B-albero vuoto con grado minimo uguale a 2. Quanti sono i nodi del B-albero che si ottengono come risultato?

19.2-5 Le foglie di un B-albero non hanno bisogno di spazio per memorizzare i puntatori ai figli, pertanto si potrebbe utilizzare un valore t più grande di quello usato per i nodi interni pur mantenendo la stessa dimensione per le pagine del disco. Si mostri come si devono modificare le procedure di creazione e inserzione in un B-albero per gestire questa situazione.

19.2-6 Supponiamo che la procedura B-Tree-Search sia realizzata in modo tale da utilizzare in ogni nodo la ricerca binaria invece della ricerca lineare. Si mostri che questa modifica permette di ottenere un tempo di CPU di $O(\lg n)$, indipendentemente dalla scelta di t in funzione del valore di n.

19.2-7 Supponiamo di avere la possibilità di scegliere in modo arbitrario la dimensione di una pagina di disco, ma che il tempo necessario per leggere una pagina del disco diventi $a + bt$, dove a e b sono delle costanti assegnate e t è il grado minimo di un B-albero che utilizza pagine della dimensione prescelta. Si descriva come scegliere

t in modo tale da minimizzare (approssimativamente) il tempo di ricerca nei B-alberi. Si determini il valore ottimo di t nel caso in cui $a = 30$ millisecondi e $b = 40$ microsecondi.

19.3 Eliminazione di una chiave da un B-albero

L'eliminazione di una chiave da un B-albero è un'operazione che ha molte caratteristiche comuni con l'operazione di inserzione di una chiave, anche se è leggermente più complicata. In questo paragrafo, invece di descrivere con precisione le istruzioni della procedura di eliminazione, descriveremo brevemente le sue caratteristiche principali.

La procedura B-TREE-DELETE ha il compito di eliminare la chiave k dal sottoalbero la cui radice è il nodo x. La procedura B-TREE-DELETE è organizzata in modo tale da essere chiamata ricorsivamente su di un nodo x solo se il numero di chiavi del nodo x è almeno uguale al grado minimo t. Si noti che questa condizione richiede almeno una chiave in più del minimo richiesto dalla definizione dei B-alberi. Pertanto può essere necessario spostare una chiave del padre nel nodo figlio prima della chiamata ricorsiva sul figlio. Questa condizione consente l'eliminazione di una chiave da un albero con un sola discesa, senza aver bisogno di tornare sui propri passi (con una eccezione che verrà descritta nel seguito). Prima di descrivere l'operazione, però, vogliamo far presente che se mai accadesse che la radice x diventasse un nodo interno senza chiavi, allora l'unico figlio di x, $c_1[x]$, diventerebbe la nuova radice e il nodo x sarebbe cancellato. Si noti che questa operazione fa diminuire di una unità l'altezza dell'albero, mentre continua a valere la proprietà che la radice dell'albero contiene sempre almeno una chiave (a meno che l'albero sia vuoto).

La figura 19.8 mostra le situazioni che si possono incontrare durante l'esecuzione dell'operazione di eliminazione di una chiave da un B-albero.

1. Se la chiave k è nel nodo x e x è una foglia, allora si elimina la chiave k da x.
2. Se la chiave k è nel nodo x e x è un nodo interno, allora si possono verificare i seguenti casi.
   a. Sia y il figlio di x che precede k. Se y ha almeno t chiavi, allora si deve trovare il predecessore di k nel sottoalbero di radice y. Sia k' il predecessore cercato. A questo punto si cancella ricorsivamente k' e poi si deve sostituire nel nodo x la chiave k con la chiave k'. (Le operazioni che trovano k' e la eliminano possono essere eseguite con una singola discesa verso le foglie.)
   b. In modo simmetrico, sia z il figlio di x che segue k. Se z ha almeno t chiavi, allora si deve trovare il successore di k nel sottoalbero di radice z. Sia k' il successore cercato. A questo punto si cancella ricorsivamente k' e poi si deve sostituire nel nodo x la chiave k con la chiave k'. (Le operazioni che trovano k' e la eliminano possono essere eseguite con una singola discesa verso le foglie.)
   c. La possibilità rimanente è che sia y che z abbiano solamente $t - 1$ chiavi. In questo caso si devono inserire nel nodo y sia la chiave k che tutte le chiavi di z (i figli di z diventano figli di y). Il primo effetto di questa operazione di fusione è che il nodo x perde sia k che il puntatore a z. Inoltre, il nodo y diventa un nodo con $2t - 1$ chiavi. A questo punto si procede ricorsivamente a eliminare k da y, dopo aver restituito la memoria occupata da z.

3. Se la chiave k non è presente nel nodo interno x, allora si deve determinare la radice $c_i[x]$ del sottoalbero che contiene la chiave k, se la chiave è presente nel B-albero. Se $c_i[x]$ ha solamente $t - 1$ chiavi, allora si esegue il passo 3a o il passo 3b per fare in modo di continuare l'operazione con un nodo che contiene almeno t chiavi. Infine si procede ricorsivamente sul sottoalbero di x prescelto.
   a. Se $c_i[x]$ ha solamente $t - 1$ chiavi ma ha un fratello con t chiavi, allora si assegna a $c_i[x]$ una chiave ulteriore prendendo una chiave dal padre x, poi si deve prendere una chiave

Figura 19.8 *L'eliminazione di chiavi da un B-albero. Il grado minimo di questo B-albero è $t = 3$, quindi un nodo (con l'eccezione della radice) non può avere meno di due chiavi. I nodi che sono modificati dalle operazioni sono leggermente più chiari degli altri nodi.* (a) *Il B-albero di figura 19.7(e).* (b) *L'eliminazione della chiave F. Questo è il caso 1: semplice eliminazione di una chiave da una foglia.* (c) *L'eliminazione della chiave M. Questo è il caso 2a: la chiave L che precede M è spostata in alto per prendere il posto di M.* (d) *L'eliminazione della chiave G. Questo è il caso 2c: la chiave G è spinta verso il basso per formare il nodo DEGJK, successivamente la chiave G è eliminata da questa foglia (caso 1).* (e) *L'eliminazione della chiave D. Questo è il caso 3b: la ricorrenza non può scendere sul nodo CL perché ha solamente 2 chiavi; pertanto la chiave P è spinta verso il basso e fusa assieme ai nodi CL e TX per formare il nodo CLPTX; successivamente la chiave D è eliminata dalla foglia (caso 1).* (e') *Dopo (d) la radice viene eliminata e l'albero si accorcia in altezza di una unità.* (f) *L'eliminazione della chiave B. Questo è il caso 3a: la chiave C è spostata e prende la posizione della chiave B, la chiave E è spostata e prende la posizione della chiave C.*

Figura 19.8 (continua)

dal fratello di destra o di sinistra di $c_i[x]$ e metterla nel padre x; infine si deve spostare il figlio appropriato dal fratello ed inserirlo in $c_i[x]$.

b. Se $c_i[x]$ e tutti i suoi fratelli hanno $t - 1$ chiavi, allora si deve fondere assieme c_i con uno dei suoi fratelli. Questa operazione comporta la discesa di una chiave di x nel nuovo nodo: questa chiave diventa la chiave mediana di quel nodo.

Nei B-alberi la maggior parte delle chiavi è situata nelle foglie, dunque capita piuttosto spesso che l'operazione di eliminazione di una chiave coinvolga solamente le foglie. La procedura B-TREE-DELETE agisce con una sola discesa nell'albero, senza aver bisogno di tornare sui propri passi. Tuttavia, quando si elimina una chiave da un nodo interno, la procedura effettua un passo in profondità ma può accadere che si debba ritornare sul nodo da cui è stata eliminata la chiave per sostituirla con la chiave precedente o seguente (casi 2a e 2b).

Anche se la procedura appare piuttosto complicata, richiede solamente $O(h)$ accessi al disco per un B-albero di altezza h. Infatti tra ogni chiamata ricorsiva e la successiva vengono eseguite solamente $O(1)$ chiamate alle operazioni DISK-READ e DISK-WRITE. Il tempo di CPU è $O(th) = O(t \log_t n)$.

Esercizi

19.3-1 Si mostri il risultato dell'eliminazione delle chiavi C, P e V, in quest'ordine, dall'albero di figura 19.8(f).

19.3-2 Si scriva in dettaglio la procedura B-TREE-DELETE.

Problemi

19.1 Pile in memoria secondaria

Consideriamo il problema di realizzare una pila in un sistema che ha a disposizione una quantità, relativamente piccola, di memoria principale ed una quantità elevata di memoria secondaria su disco. Supponiamo che i valori passati o restituiti dalle operazioni PUSH e POP siano costituiti da una parola di memoria. Infine, supponiamo che la pila cresca a tal punto da non poter risiedere completamente nella memoria principale, e così la maggior parte della pila è memorizzata nella memoria secondaria.

La più semplice realizzazione, anche se non particolarmente efficiente, della pila è quella che la fa risiedere completamente nella memoria secondaria. In questa realizzazione, si mantiene nella memoria principale un puntatore alla testa della pila e questo puntatore non è altro che l'indirizzo nel disco dell'elemento in testa alla pila. Se indichiamo con p il valore del puntatore, l'elemento in testa alla pila è la parola numero $p \bmod m$ della pagina di disco $\lfloor p/m \rfloor$, dove m è il numero di parole contenute in una pagina.

Per realizzare l'operazione PUSH basta incrementare il puntatore alla testa della pila, leggere nel disco la pagina corrispondente al puntatore e portarla in memoria principale, copiare il parametro dell'operazione PUSH nella opportuna parola all'interno della pagina e scrivere la pagina modificata nel disco. L'operazione POP è simile: si decrementa il puntatore alla testa della pila, si legge dal disco la pagina corrispondente al puntatore e si restituisce l'elemento in testa alla pila. In questo caso non si deve riscrivere la pagina nel disco visto che la pagina non è modificata.

Dato che le operazioni sul disco sono relativamente costose, per avere un'idea dell'efficienza della realizzazione basta considerare il numero di accessi al disco. Considereremo anche il tempo di CPU, ma addebitiamo costo $\Theta(m)$ per ogni accesso sul disco a una pagina con m parole.

a. Nel caso pessimo, qual è il numero di accessi al disco per una sequenza di n operazioni della pila basate su questa semplice realizzazione? Qual è il tempo di CPU per una sequenza di n operazioni? (Sia in questo caso, che nel seguito, la risposta dev'essere una espressione che dipende da m e n.)

Ora adottiamo una diversa strategia per realizzare la pila. Questa nuova strategia è caratterizzata dal fatto che una pagina della pila risiede sempre in memoria principale. (In memoria principale viene anche memorizzata l'informazione su quale sia la pagina corrente, cioè la pagina che in quel momento risiede in memoria.) Possiamo eseguire una qualunque delle operazioni della pila solo se la pagina della pila coinvolta nell'operazione risiede in memoria. Se necessario, la pagina corrente è scritta nel disco ed una nuova pagina può essere trasferita dal disco alla memoria. Si noti che se la pagina coinvolta nell'operazione è quella corrente, allora non c'è bisogno di accedere al disco.

b. Qual è nel caso pessimo il numero di accessi al disco necessari per eseguire una sequenza di n operazioni PUSH? Qual è il tempo di CPU?

c. Qual è nel caso pessimo il numero di accessi al disco necessari per eseguire una sequenza di n operazioni della pila? Qual è il tempo di CPU?

Supponiamo ora di realizzare la pila tenendo in memoria principale due pagine (oltre alla memoria necessaria per tenere traccia delle pagine residenti in memoria).

d. Si descriva una modalità di gestione delle pagine della pila tale che, per ogni operazione della pila, il numero ammortizzato di accessi al disco sia $O(1/m)$ e il tempo di CPU ammortizzato sia $O(1)$.

19.2 *Unione e divisione di alberi 2-3-4*

L'operazione ***join*** prende due insiemi dinamici S' e S'' e un elemento x tale che, per ogni $x' \in S'$ e $x'' \in S''$, abbiamo che $key[x'] < key[x] < key[x'']$. Questa operazione restituisce come risultato l'insieme $S = S' \cup \{x\} \cup S''$. L'operazione ***split*** è sostanzialmente l'inversa dell'operazione join: dato un insieme dinamico S e un elemento x in S, l'operazione crea un insieme S' che consiste di tutti gli elementi che appartengono a $S - \{x\}$ le cui chiavi sono più piccole di $key[x]$, e un insieme S'' che consiste di tutti gli elementi che appartengono a $S - \{x\}$ le cui chiavi sono più grandi di $key[x]$. In questo problema ci domandiamo come realizzare queste due operazioni su alberi 2-3-4. Per semplicità assumiamo che gli elementi coincidano con le chiavi e che tutte le chiavi siano distinte.

a. Per ogni nodo x di un albero 2-3-4, si mostri come sia possibile avere un campo $height[x]$, che contenga l'altezza dell'albero radicato al nodo x. Bisogna assicurare che la soluzione proposta non influenzi il tempo di esecuzione asintotico delle operazioni di ricerca, inserzione ed eliminazione.

b. Si mostri come realizzare l'operazione join. Siano T' e T'' due alberi 2-3-4 e sia k una chiave: l'operazione join deve essere eseguita in tempo $O(|h' - h''|)$, dove h' e h'' sono le altezze di T' e T'', rispettivamente.

c. Sia T un albero 2-3-4 e p un cammino dalla radice dell'albero a una determinata chiave k. Inoltre sia S' l'insieme delle chiavi di T che sono minori di k, e sia S'' l'insieme delle chiavi di T che sono maggiori della chiave k. Si mostri che il cammino p divide S' in un insieme di alberi $\{T'_0, T'_1, \ldots, T'_m\}$ e in un insieme di chiavi $\{k'_1, k'_2, \ldots, k'_m\}$ dove, per ogni $i = 1, 2, \ldots, m$, abbiamo che $y < k'_i < z$ per ogni chiave $y \in T'_{i-1}$ e $z \in T'_i$. Qual è la relazione esistente tra le altezze di T'_{i-1} e di T'_i? Si descriva come il cammino p divide l'insieme S'' in insiemi di alberi e chiavi.

d. Si mostri come realizzare l'operazione split su un albero 2-3-4 T. Si usi l'operazione join per mettere le chiavi di S' in un unico albero 2-3-4 e le chiavi di S'' in un unico albero 2-3-4. Il tempo di esecuzione dell'operazione split deve essere $O(\lg n)$, dove n è il numero delle chiavi di T. (*Suggerimento*: i costi per l'unione dovrebbero dare una somma telescopica, come in (3.7).)

Note al capitolo

Eccellenti riferimenti per gli schemi su alberi bilanciati e B-alberi sono Knuth [123], Aho, Hopcroft e Ullman [4], Sedgewick [175]. Comer [48] presenta una rassegna completa sui B-alberi. Guibas e Sedgewick [93] studiano le relazioni esistenti tra i vari tipi di alberi bilanciati, compresi gli alberi RB e gli alberi 2-3-4.

Nel 1970 J. E. Hopcroft ha introdotto gli alberi 2-3, i precursori dei B-alberi e degli alberi 2-3-4. Questi alberi hanno la caratteristica che ogni nodo interno possiede due o tre figli. I B-alberi sono stati introdotti da Bayer e McCreight nel 1972 [18]; gli autori non hanno mai spiegato l'origine del nome.

Heap binomiali

20

Questo capitolo ed il Capitolo 21 presentano una classe di strutture di dati conosciute con il nome di ***heap aggregabili***. Queste strutture di dati forniscono le cinque operazioni seguenti.

MAKE-HEAP() crea e restituisce come risultato un nuovo heap che non contiene nessun elemento.

INSERT(H, x) inserisce un nodo x, la cui chiave è già stata definita, nello heap H.

MINIMUM(H) restituisce come risultato un puntatore al nodo dello heap la cui chiave è minima.

EXTRACT-MIN(H) elimina dallo heap H l'elemento di chiave minima e restituisce come risultato il puntatore a quel nodo.

UNION(H_1, H_2) crea e restituisce come risultato un nuovo heap che contiene tutti i nodi degli heap H_1 e H_2. Come effetto di questa operazione gli heap H_1 e H_2 sono distrutti.

Inoltre, le strutture di dati che vengono esaminate in questi capitoli forniscono anche le due operazioni seguenti.

DECREASE-KEY(H, x, k) assegna al nodo x dello heap H una nuova chiave k. Il valore della nuova chiave non deve essere più grande del vecchio valore.

DELETE(H, x) elimina il nodo x dallo heap H.

Se non si considera l'operazione UNION, gli heap binari ordinari, definiti nel Capitolo 7, sono piuttosto efficienti, come è descritto dalla tabella mostrata nella figura 20.1. Nel caso degli heap binari tutte le operazioni, con la sola eccezione dell'operazione UNION, hanno tempo di esecuzione $O(\lg n)$ nel caso pessimo (in alcuni casi il tempo di esecuzione è addirittura migliore). Tuttavia le prestazioni degradano notevolmente se si considera anche l'operazione UNION. Infatti la concatenazione dei due array che contengono i due heap da unire, e successivamente l'esecuzione dell'operazione HEAPIFY, portano a un tempo di esecuzione dell'operazione UNION di $\Theta(n)$ nel caso pessimo.

In questo capitolo esamineremo gli "heap binomiali": i tempi di esecuzione delle operazioni (nel caso pessimo) applicate agli heap binomiali sono riportati nella tabella della figura 20.1. In particolare, per fondere assieme due heap binomiali con un totale di n elementi, l'operazione UNION richiede solamente tempo $O(\lg n)$.

Nel Capitolo 21 esamineremo le caratteristiche degli heap di Fibonacci, che per alcune operazioni presentano limiti di tempo migliori. Tuttavia, vogliamo fare notare che i tempi di

Procedura	Heap binario (caso pessimo)	Heap binomiale (caso pessimo)	Heap di Fibonacci (ammortizzato)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Figura 20.1 *Il tempo di esecuzione delle operazioni nei tre casi di heap aggregabili. Con n si indica il numero degli elementi negli heap al momento della chiamata delle operazioni.*

esecuzione delle operazioni per gli heap di Fibonacci riportati nella figura 20.1 sono limiti di tempo ammortizzati e non limiti di tempo nel caso pessimo.

In questo capitolo non vengono prese in esame tutte quelle problematiche relative all'assegnamento della memoria prima dell'inserimento di un nodo nello heap, e alla restituzione della memoria dopo l'esecuzione di una operazione di eliminazione. Assumiamo che questi problemi siano gestiti dalle procedure dello heap.

L'operazione di ricerca non è particolarmente efficiente negli heap binari: trovare un nodo con una determinata chiave può richiedere parecchio tempo. La medesima considerazione vale anche per gli heap binomiali e per gli heap di Fibonacci. Per questo motivo tutte le operazioni che fanno riferimento a un determinato nodo (per esempio DECREASE-KEY o DELETE) richiedono come parametro di ingresso il puntatore a quel nodo. In molte applicazioni questo vincolo non comporta problemi particolari.

Il paragrafo 20.1, dopo aver introdotto gli alberi binomiali, definisce gli heap binomiali. Viene anche introdotta una rappresentazione specifica degli heap binomiali. Il paragrafo 20.2 mostra come sia possibile realizzare le operazioni degli heap binomiali con i limiti di tempo riportati nella figura 20.1.

20.1 Alberi binomiali e heap binomiali

Uno heap binomiale è un insieme di alberi binomiali, quindi in questo paragrafo per prima cosa definiamo gli alberi binomiali e ne dimostriamo alcune proprietà di base. Successivamente, introduciamo gli heap binomiali ed una loro rappresentazione.

20.1.1 Alberi binomiali

Un ***albero binomiale*** B_k è un albero ordinato (si veda il paragrafo 5.5.2) definito ricorsivamente. La figura 20.2(a) mostra un albero binomiale B_0 che consiste di un solo nodo. L'albero binomiale B_k consiste di due alberi binomiali B_{k-1} ***collegati*** assieme: la radice di uno dei due alberi è il figlio più a sinistra della radice dell'altro. La figura 20.2(b) descrive alcuni alberi binomiali: gli alberi binomiali B_0, B_1, B_2, B_3 e B_4.

Il seguente lemma definisce alcune proprietà degli alberi binomiali.

Figura 20.2 **(a)** *La definizione ricorsiva dell'albero binomiale B_k. I triangoli rappresentano sottoalberi con radice.* **(b)** *Alberi binomiali da B_0 a B_4. Viene mostrata la profondità dei nodi dell'albero B_4.* **(c)** *Un modo alternativo di considerare l'albero binomiale B_k.*

Lemma 20.1 (Proprietà di alberi binomiali)

Sia B_k un albero binomiale: allora valgono le seguenti proprietà:

1. i nodi dell'albero sono 2^k,
2. l'altezza dell'albero è k,
3. i nodi a profondità i sono esattamente $\binom{k}{i}$, per $i = 0, 1, \ldots, k$, e
4. la radice dell'albero ha grado k. Il grado della radice è maggiore del grado di ogni altro nodo; inoltre se $k-1, k-2, \ldots, 0$ è una enumerazione, da sinistra verso destra, dei figli della radice, allora il figlio numero i è la radice di un sottoalbero binomiale B_i.

Dimostrazione. La dimostrazione è per induzione su k. Per ognuna delle proprietà la base dell'induzione è l'albero binomiale B_0. La verifica della validità delle proprietà per l'albero binomiale B_0 è immediata.

Per dimostrare i passi induttivi, assumiamo che il lemma sia verificato per l'albero binomiale B_{k-1}.

1. L'albero binomiale B_k consiste di due copie di B_{k-1}, quindi B_k ha $2^{k-1} + 2^{k-1} = 2^k$ nodi.
2. L'albero B_k è costituito da due copie dell'albero B_{k-1}, collegate assieme, pertanto la profondità massima di un nodo nell'albero B_k si ottiene sommando 1 alla profondità

massima di B_{k-1}. Per l'ipotesi induttiva, la massima profondità dell'albero B_k risulta essere $(k-1)+1=k$.

3. Sia $D(k, i)$ il numero dei nodi a profondità i dell'albero binomiale B_k. Per costruzione l'albero B_k è costituito da due copie dell'albero B_{k-1} collegate assieme, pertanto un nodo a profondità i in B_{k-1} nell'albero B_k appare una volta a profondità i e l'altra volta a profondità $i+1$. In altri termini, abbiamo che il numero dei nodi a profondità i nell'albero B_k è dato dalla somma del numero dei nodi a profondità i di B_{k-1} con il numero dei nodi a profondità $i-1$ in B_{k-1}. Quindi

$$\begin{aligned} D(k,i) &= D(k-1,i)+D(k-1,i-1) \\ &= \binom{k-1}{i}+\binom{k-1}{i-1} \\ &= \binom{k}{i}. \end{aligned}$$

La seconda uguaglianza segue direttamente dall'ipotesi induttiva, e la terza uguaglianza segue dall'Esercizio 6.1-7.

4. Il solo nodo con un grado più alto in B_k rispetto a quello che aveva in B_{k-1} è il nodo radice che ha un figlio in più di quanti ne aveva in B_{k-1}. Dato che la radice di B_{k-1} ha grado $k-1$, segue che la radice di B_k ha grado k. Per l'ipotesi induttiva, e come è mostrato dall'esempio di figura 20.2(c), i figli della radice di B_{k-1} sono, da sinistra a destra, le radici di $B_{k-2}, B_{k-3}, \ldots, B_0$. Pertanto, quando si collegano assieme i due alberi B_{k-1} abbiamo che i figli della radice risultante diventano $B_{k-1}, B_{k-2}, \ldots, B_0$. ■

Corollario 20.2

Il massimo grado di ogni nodo in un albero binomiale con n nodi è $\lg n$.

Dimostrazione. È una immediata conseguenza delle proprietà 1 e 4 del lemma 20.1. ■

Il termine "albero binomiale" deriva dalla proprietà 3 del lemma 20.1: i termini $\binom{k}{i}$ sono i coefficienti binomiali. L'Esercizio 20.1-3 fornisce ulteriori giustificazioni per la scelta del termine "binomiale".

20.1.2 Heap binomiali

Uno ***heap binomiale*** H è un insieme di alberi binomiali che soddisfa le seguenti proprietà dette ***proprietà dello heap binomiale***.

1. Ogni albero binomiale in H ha la proprietà di ***ordinamento parziale dello heap***: la chiave di un nodo è maggiore o uguale della chiave del nodo padre.
2. Non esistono due alberi binomiali in H le cui radici hanno lo stesso grado.

La prima proprietà garantisce che la chiave della radice di uno heap binomiale sia la più piccola chiave dello heap.

La seconda proprietà implica che uno heap binomiale con n nodi consiste al massimo di $\lfloor \lg n \rfloor + 1$ alberi binomiali. Infatti si osservi come la rappresentazione binaria del numero n contenga un numero di bit uguale a $\lfloor \lg n \rfloor + 1$; supponiamo che questi bit siano $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \ldots, b_0 \rangle$ e quindi

Figura 20.3 *Uno heap binomiale H con $n = 13$ nodi.* **(a)** *Lo heap è costituito dall'insieme di alberi binomiali B_0, B_2 e B_3, rispettivamente con 1, 4 e 8 nodi, per un totale di 13 nodi. Ogni albero binomiale rispetta l'ordinamento dello heap, quindi la chiave di un nodo qualunque non è più piccola della chiave del nodo padre. La figura mostra anche la lista delle radici, che contiene tutte le radici ordinate per grado crescente.* **(b)** *Una rappresentazione più dettagliata dello heap binomiale H. Ogni albero binomiale è memorizzato usando la rappresentazione figlio-sinistro, fratello-destro; inoltre in ogni nodo viene memorizzato il grado del nodo.*

$n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i \cdot 2^i$. Per la proprietà 1 del lemma 20.1 abbiamo che l'albero binomiale B_i è presente in H se e solo se $b_i = 1$. Pertanto, lo heap binomiale H contiene al massimo $\lfloor \lg n \rfloor + 1$ alberi binomiali.

La figura 20.3(a) descrive uno heap binomiale H con 13 nodi. Poiché la rappresentazione binaria di 13 è $\langle 1101 \rangle$, H è costituito dall'insieme di alberi binomiali con l'ordinamento parziale dello heap B_3, B_2 e B_0. L'albero binomiale B_3 ha 8 nodi, B_2 ha 4 nodi e B_0 ha un solo nodo, pertanto il numero totale dei nodi dello heap è 13.

Una rappresentazione per gli heap binomiali

Come appare chiaramente dalla figura 20.3(b), ogni albero binomiale è memorizzato con la rappresentazione figlio-sinistro, fratello-destro descritta nel paragrafo 11.4. Ogni nodo è caratterizzato da un insieme di informazioni. Oltre al campo chiave, *key*, sono presenti informazioni satellite che dipendono dall'applicazione corrente. Inoltre, ogni nodo x contiene un certo numero di puntatori: il puntatore $p[x]$ al nodo padre, il puntatore $child[x]$ al suo figlio

Figura 20.4 *L'albero binomiale B_4 i cui nodi sono etichettati con numeri binari corrispondenti all'ordine di una visita differita.*

più a sinistra e il puntatore *sibling*[x] al primo fratello di destra del nodo x. Se x è una radice, allora $p[x]$ = NIL. Se il nodo x non ha figli, allora *child*[x] = NIL; inoltre se x è il figlio più a destra, allora *sibling*[x] = NIL. Infine, ogni nodo x contiene un ulteriore campo, *degree*[x], per indicare il numero dei suoi figli: il grado del nodo.

La figura 20.3 mostra anche un'altra caratteristica della rappresentazione degli heap binomiali: le radici degli alberi binomiali dello heap sono organizzate in una lista. Spesso, faremo riferimento a questa lista con il nome di ***lista delle radici***. Si noti che le radici nella lista delle radici sono ordinate in modo crescente rispetto al grado; inoltre, per la seconda proprietà degli heap binomiali abbiamo che, in un qualunque heap binomiale con n nodi, i gradi delle radici costituiscono un sottoinsieme di $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$. Nel caso delle radici, il campo *sibling* ha un ruolo differente rispetto a quello che aveva nei nodi interni. Infatti, se x è una radice, allora *sibling*[x] è il puntatore alla prossima radice nella lista delle radici. (Chiaramente, *sibling*[x] = NIL se x è l'ultimo elemento della lista delle radici).

Il puntatore *head*[H] fornisce il meccanismo di accesso allo heap binomiale H. Si noti che *head*[H] è il puntatore alla testa della lista delle radici di H, inoltre *head*[H] = NIL se lo heap binomiale non contiene elementi.

Esercizi

20.1-1 Supponiamo che x sia un nodo di un albero binomiale contenuto in uno heap binomiale e che *sibling*[x] ≠ NIL. Se x non è una radice, qual è la relazione esistente tra *degree*[*sibling*[x]] e *degree*[x]? Cosa succede quando x è una radice?

20.1-2 Sia x un nodo, diverso da un nodo radice, di un albero binomiale contenuto in uno heap binomiale: qual è la relazione esistente tra *degree*[$p[x]$] e *degree*[x]?

20.1-3 Supponiamo di associare delle etichette binarie ai nodi di un albero binomiale B_k mediante una visita in ordine differito, come descritto nella figura 20.4. Sia x un nodo a profondità i e sia l la sua etichetta; inoltre, sia $j = k - i$. Si dimostri che x ha esattamente j bit uguali a 1 nella sua rappresentazione binaria. Quante stringhe binarie con k bit contengono esattamente j bit uguali a 1? Si dimostri che il grado del nodo x è uguale al numero di 1 che stanno alla destra dello 0 più a destra nella rappresentazione binaria di l.

20.2 Operazioni su heap binomiali

In questo paragrafo mostreremo come sia possibile realizzare le operazioni su heap binomiali nei limiti di tempo descritti dalla figura 20.1. Prenderemo in esame solamente i limiti superiori delle operazioni; l'esame dei limiti inferiori è lasciato per esercizio (Esercizio 20.2-10).

Creazione di un nuovo heap binomiale

La procedura MAKE-BINOMIAL-HEAP crea uno heap binomiale vuoto: la procedura restituisce un oggetto H, con *head*[H] = NIL, dopo avergli assegnato lo spazio di memoria opportuno. Chiaramente il tempo di esecuzione è $\Theta(1)$.

Ricerca della chiave minima

La procedura BINOMIAL-HEAP-MINIMUM ha come parametro di ingresso uno heap binomiale H con n nodi, e restituisce come risultato il puntatore al nodo con la chiave minima. La realizzazione che andiamo a esaminare assume che non esistano chiavi il cui valore sia ∞. (A questo proposito si consideri l'Esercizio 20.2-5.)

```
BINOMIAL-HEAP-MINIMUM(H)
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL
      do if key[x] < min
            then min ← key[x]
                 y ← x
         x ← sibling[x]
return y
```

Uno heap binomiale soddisfa la proprietà dell'ordinamento heap e, pertanto, la chiave minima deve necessariamente essere associata a un nodo radice. La procedura BINOMIAL-HEAP-MINIMUM esamina tutte le radici, il cui numero è al massimo $\lfloor \lg n \rfloor + 1$, e memorizza nella variabile *min* il minimo corrente e nella variabile y il puntatore al minimo corrente. Per esempio, la chiamata della procedura BINOMIAL-HEAP-MINIMUM, con argomento lo heap descritto nella figura 20.3, restituisce come risultato un puntatore al nodo la cui chiave ha il valore 1.

Poiché la procedura BINOMIAL-HEAP-MINIMUM esamina al massimo $\lfloor \lg n \rfloor + 1$ radici, il suo tempo di esecuzione è $O(\lg n)$.

Unione di due heap binomiali

L'operazione che unisce due heap binomiali verrà utilizzata come sottoprogramma dalla maggior parte delle operazioni che presenteremo nel seguito. La procedura BINOMIAL-HEAP-UNION collega assieme tutti gli alberi binomiali le cui radici hanno lo stesso grado. La seguente procedura collega assieme l'albero B_{k-1} di radice y con l'albero B_{k-1} di radice z: ovvero z diviene il padre di y. Pertanto z diviene la radice di un albero B_k.

```
BINOMIAL-LINK (y, z)
  p[y] ← z
  sibling[y] ← child[z]
  child[z] ← y
  degree[z] ← degree[z] + 1
```

La procedura BINOMIAL-LINK inserisce il nodo y in testa alla lista dei figli del nodo z in tempo $O(1)$. La procedura opera correttamente dato che la rappresentazione figlio-sinistro, fratello-destro di ogni heap binomiale soddisfa l'ordinamento degli alberi nello heap: in un albero B_k il figlio più a sinistra della radice è la radice di un albero B_{k-1}.

La procedura seguente unisce due heap binomiali H_1 e H_2 e restituisce lo heap risultato dell'unione. Durante il calcolo la procedura distrugge la rappresentazione di H_1 e H_2. Oltre alla procedura BINOMIAL-LINK, la procedura si avvale della procedura ausiliaria BINOMIAL-HEAP-MERGE che fonde assieme in un'unica lista le liste delle radici di H_1 e H_2. Questa nuova lista è ordinata in modo crescente rispetto al grado dei nodi. La realizzazione puntuale della procedura BINOMIAL-HEAP-MERGE è lasciata per esercizio (Esercizio 20.2-2): la procedura è simile alla procedura MERGE definita nel paragrafo 1.3.1.

```
BINOMIAL-HEAP-UNION(H1, H2)
 1  H ← MAKE-BINOMIAL-HEAP()
 2  head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
 3  libera gli oggetti H1 e H2 ma non le liste cui si riferiscono
 4  if head[H] = NIL
 5     then return H
 6  prev-x ← NIL
 7  x ← head[H]
 8  next-x ← sibling[x]
 9  while next-x ≠ NIL
10     do if degree[x] ≠ degree[next-x] oppure
                 (sibling[next-x] ≠ NIL
                 e degree[sibling[next-x]] = degree[x])
11           then prev-x ← x                                  ▷ Casi 1 e 2
12                x ← next-x                                  ▷ Casi 1 e 2
13           else if key[x] ≤ key[next-x]
```



```
14                then sibling[x] ← sibling[next-x]           ▷ Caso 3
15                     BINOMIAL-LINK (next-x, x)               ▷ Caso 3
16                else if prev-x = NIL                         ▷ Caso 4
17                        then head[H] ← next-x                ▷ Caso 4
18                        else sibling[prev-x] ← next-x        ▷ Caso 4
19                     BINOMIAL-LINK (x, next-x)               ▷ Caso 4
20                     x ← next-x                              ▷ Caso 4
21        next-x ← sibling[x]
22  return H
```

La figura 20.5 mostra un esempio della procedura BINOMIAL-HEAP-UNION in cui è presente ciascuno dei quattro casi che si possono manifestare.

Concettualmente la procedura BINOMIAL-HEAP-UNION è divisa in due fasi: la prima fase, costituita dalla chiamata della procedura BINOMIAL-HEAP-MERGE, fonde assieme in un'unica lista, ordinata in modo crescente rispetto al grado dei nodi, le liste delle radici degli heap binomiali H_1 e H_2. Potrebbe accadere che esistano due radici (ma non più di due) con lo stesso grado: per questo motivo la seconda fase collega assieme tutte le radici con lo stesso grado: pertanto quando la seconda fase è terminata non esistono più due radici con lo stesso grado. Tutte le operazioni risultano molto efficienti dato che la lista H è una lista ordinata per gradi crescenti.

A questo punto esaminiamo in dettaglio le caratteristiche della procedura. Prima di tutto si fondono assieme in una unica lista le liste delle radici di H_1 e H_2 (linee 1-3). Le liste delle radici di H_1 e H_2, sono ordinate per gradi crescenti e la procedura BINOMIAL-HEAP-MERGE restituisce una lista ordinata per gradi crescenti. Se m è il numero complessivo degli elementi delle liste delle radici di H_1 e H_2 allora BINOMIAL-HEAP-MERGE richiede tempo $O(m)$ dato che la procedura esamina le radici in testa alle liste delle radici di H_1 e H_2, successivamente concatena la radice con il grado più basso nella lista delle radici risultato ed allo stesso tempo la rimuove dalla relativa lista in ingresso.

Se i due heap binomiali in ingresso sono entrambi vuoti la procedura termina (linee 4-5): l'unione di due heap vuoti è lo heap vuoto. Si noti che a partire dalla linea 6 sicuramente la lista H contiene almeno un elemento. A questo punto la procedura BINOMIAL-HEAP-UNION introduce alcuni puntatori alla lista delle radici di H:

- x è il puntatore alla radice in esame,
- *prev-x* è il puntatore alla radice che precede x nella lista delle radici: *sibling*[*prev-x*] = x, e
- *next-x* è il puntatore alla radice che segue x nella lista delle radici: *sibling*[x] = *next-x*.

Dato che H_1 e H_2 sono heap binomiali, sia la lista delle radici di H_1 che quella di H_2 non possono contenere due nodi con lo stesso grado; però la lista H che le unisce può contenere due nodi con lo stesso grado. Tuttavia, la struttura della procedura BINOMIAL-HEAP-MERGE garantisce che le radici con lo stesso grado siano nodi adiacenti nella lista H.

Durante l'esecuzione della procedura BINOMIAL-HEAP-UNION può accadere che la lista H, a un determinato istante, contenga tre radici con lo stesso grado. Vedremo tra un attimo come questa situazione possa manifestarsi. A ogni iterazione del ciclo **while** (linee 9-21) in base al grado di x e di *next-x*, e anche in base al grado di *sibling*[*next-x*], si decide se collegare x

Figura 20.5 *L'esecuzione della procedura* BINOMIAL-HEAP-UNION. (a) *Gli heap binomiali H_1 e H_2.* (b) *Lo heap binomiale H è il risultato dell'esecuzione della chiamata* BINOMIAL-HEAP-MERGE(H_1, H_2). *Inizialmente, x è la prima radice nella lista delle radici H. Dato che sia x che next-x hanno grado 0 e che key[x]< key[next-x] siamo nel caso 3.* (c) *Dopo aver collegato assieme le radici abbiamo che x è la prima di tre radici con lo stesso grado: caso 2.* (d) *Alla fine i puntatori sono avanzati di una posizione nella lista delle radici: inoltre x è la prima di due radici con lo stesso grado: caso 4.* (e) *Dopo aver collegato assieme le radici si presenta il caso 3.* (f) *Dopo un ulteriore collegamento tra le radici si presenta il caso 1: il grado di x è 3, il grado di next-x è 4. Questa è l'ultima iterazione del ciclo* while: *dopo aver eseguito il corpo del ciclo i puntatori vengono avanzati di una posizione nella lista delle radici e pertanto next-x =* NIL.

e *next-x*. Una proprietà invariante del ciclo è che, ogni volta che si comincia a eseguire il corpo del ciclo, sia *x* che *next-x* sono diversi da NIL.

Il caso 1, mostrato nella figura 20.6(a), si presenta quando $degree[x] \neq degree[next\text{-}x]$, ovvero quando *x* è la radice di un albero B_k e *next-x* è la radice di un albero B_l per qualche $l > k$. Le istruzioni delle linee 11-12 hanno il compito di gestire questa situazione: i nodi *x* e *next-x* non vengono collegati ed i puntatori sono fatti avanzare di una posizione nella lista *H*. La modifica del puntatore *next-x* (linea 21) è un'istruzione comune a tutti i casi: a seguito della modifica il puntatore *next-x* punta al nodo che segue il nuovo nodo *x*.

Il caso 2, mostrato nella figura 20.6(b), si presenta quando *x* è la prima di tre radici con lo stesso grado, cioè quando

$$degree[x] = degree[next\text{-}x] = degree[sibling[next\text{-}x]].$$

Figura 20.5 (*continua*)

Questa situazione viene affrontata in modo analogo al caso 1: i puntatori sono fatti avanzare di una posizione nella lista *H*. L'istruzione della linea 10 fa i controlli necessari sia per il caso 1 che per il caso 2; infine le istruzioni delle linee 11-12 hanno il compito di gestire entrambi i casi.

I casi 3 e 4 si presentano quando *x* è la prima di due radici con lo stesso grado:

$$degree[x] = degree[next\text{-}x] \neq degree[sibling[next\text{-}x]].$$

Questi due casi si possono presentare al passo di iterazione che segue un caso qualsiasi e, sicuramente, uno dei due casi si presenta sempre dopo il caso 2. I due casi sono sostanzialmente simmetrici e dipendono dal controllo di chi tra *x* e *next-x* ha la chiave più piccola: questo controllo stabilisce quale sia il nodo che diventerà il nodo radice dopo che i due nodi sono stati collegati assieme.

Nel caso 3, mostrato nella figura 20.6(c), $key[x] \leq key[next\text{-}x]$, e quindi *next-x* è collegato a *x*. L'istruzione della linea 14 rimuove *next-x* dalla lista delle radici e l'istruzione della linea 15 fa diventare *next-x* il figlio più a sinistra di *x*.

Nel caso 4, mostrato nella figura 20.6(d), *next-x* ha la chiave più piccola, e quindi *x* è collegato a *next-x*. Le istruzioni delle linee 16-18 eliminano *x* dalla lista delle radici; nell'esecuzione di questa istruzione si possono presentare due casi: *x* è la prima radice nella lista (linea 17), oppure *x* non è la prima radice della lista (linea 18). Infine l'istruzione della linea 19 fa diventare *x* il figlio più a sinistra di *next-x*, e l'istruzione della linea 20 aggiorna il valore di *x* per l'iterazione successiva.

Figura 20.6 *I quattro casi che si presentano durante l'esecuzione della procedura* BINOMIAL-HEAP-UNION. *Le etichette* a, b, c e d *sono utilizzate per identificare le radici coinvolte nelle operazioni; queste etichette non indicano né il grado né le chiavi associate ai nodi. In tutti i casi esaminati x è radice di un albero B_k e $l > k$.* (a) *Caso 1: degree[x] ≠ degree[next-x]. I puntatori vengono avanzati di una posizione nella lista delle radici.* (b) *Caso 2: degree[x] = degree[next-x] = degree[sibling[next-x]]. Anche in questo caso i puntatori vengono avanzati di una posizione nella lista delle radici e alla successiva iterazione vengono eseguite le istruzioni del caso 3 o del caso 4.* (c) *Caso 3: degree[x] = degree[next-x] ≠ degree[sibling[next-x]] e key[x] ≤ key[next-x]. Si elimina next-x dalla lista delle radici e lo si collega a x, creando in questo modo un albero B_{k+1}.* (d) *Caso 4: degree[x] = degree[next-x] ≠ degree[sibling[next-x]] e key[next-x] < key[x]. Si elimina x dalla lista delle radici e lo si collega a next-x: di nuovo questa operazione crea un albero B_{k+1}.*

Dopo aver eseguito le istruzioni del caso 3 o del caso 4 l'iterazione successiva del ciclo **while** viene affrontata dallo stesso stato: il puntatore x punta all'albero B_{k+1} ottenuto collegando assieme due alberi B_k. Nella lista delle radici restituita come risultato dalla procedura BINOMIAL-HEAP-MERGE potevano essere presenti zero, uno o due alberi B_{k+1}, quindi x punta al primo di uno, due o tre alberi B_{k+1}. Se x è il solo albero B_{k+1}, alla successiva iterazione vengono eseguite le istruzioni del caso 1: $degree[x] \neq degree[next\text{-}x]$. Se x è il primo di due alberi B_{k+1}, allora alla successiva iterazione vengono eseguite le istruzioni del caso 3 o del caso 4. Infine, se x è il primo di tre alberi B_{k+1} allora alla successiva iterazione vengono eseguite le istruzioni del caso 2.

Il tempo di esecuzione della procedura BINOMIAL-HEAP-UNION è $O(\lg n)$, dove n è il numero complessivo di nodi degli heap binomiali H_1 e H_2. Per dimostrare l'affermazione precedente, sia n_1 il numero di nodi di H_1 e sia n_2 il numero di nodi di H_2: $n = n_1 + n_2$. Abbiamo allora che H_1

contiene al massimo un numero di radici pari a $\lfloor \lg n_1 \rfloor + 1$ e H_2 contiene al massimo $\lfloor \lg n_2 \rfloor + 1$ radici; quindi, dopo la chiamata della procedura BINOMIAL-HEAP-MERGE, H contiene al massimo un numero di radici pari a $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq 2\lfloor \lg n \rfloor + 2 = O(\lg n)$. Il tempo di esecuzione della procedura BINOMIAL-HEAP-MERGE è pertanto $O(\lg n)$. Ogni iterazione del ciclo **while** viene eseguita in tempo $O(1)$ e chiaramente il numero massimo di iterazioni è $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$: a ogni iterazione il puntatore viene fatto avanzare di una posizione nella lista delle radici H o si elimina una radice dalla lista delle radici. Pertanto, il tempo totale è $O(\lg n)$.

Inserimento di un nodo

La seguente procedura inserisce un nodo x in uno heap binomiale H. Naturalmente assumiamo che il nodo x abbia già un suo spazio di memoria e che la sua chiave $key[x]$ sia già stata definita.

```
BINOMIAL-HEAP-INSERT(H, x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H, H')
```

La procedura crea in tempo $O(1)$ uno heap binomiale H' con un solo nodo; successivamente unisce questo heap con lo heap H in tempo $O(\lg n)$. La chiamata della procedura BINOMIAL-HEAP-UNION ha anche il compito di liberare la posizione di memoria assegnata allo heap temporaneo H'. (Una realizzazione della procedura che non fa uso della procedura BINOMIAL-HEAP-UNION è descritta nell'Esercizio 20.2-8.)

Estrazione del nodo con la chiave minima

La seguente procedura elimina da uno heap binomiale H il nodo con la chiave minima e restituisce come risultato un puntatore a quel nodo.

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1  trova la radice x con la chiave minima nella lista delle radici di H ed elimina x dalla lista
       delle radici di H
2  H' ← MAKE-BINOMIAL-HEAP()
3  inverti l'ordine della lista dei figli di x e assegna a head[H'] il puntatore al primo
       elemento della lista risultante
4  H ← BINOMIAL-HEAP-UNION(H, H')
5  return x
```

Figura 20.7 Il *risultato dell'esecuzione della procedura* BINOMIAL-HEAP-EXTRACT-MIN. (a) *Uno heap binomiale H.* (b) *La radice x con la chiave minima viene eliminata dalla lista delle radici di H.* (c) *La lista dei figli di x viene invertita; come risultato si ottiene un altro heap binomiale H'.* (d) *Il risultato dell'unione di H e H'.*

Il comportamento di questa procedura è mostrato nella figura 20.7. Lo heap binomiale in ingresso H è descritto nella figura 20.7(a). La figura 20.7(b) mostra la situazione che si viene a creare dopo l'esecuzione dell'istruzione della linea 1: la radice x con la chiave minima è eliminata dalla lista delle radici di H. Nel caso in cui x sia la radice di un albero B_k, allora per la proprietà 4 del lemma 20.1 abbiamo che i figli del nodo x da sinistra verso destra sono le radici di alberi $B_{k-1}, B_{k-2}, \ldots, B_0$. La figura 20.7(c) fa vedere come l'operazione della riga 3, che inverte la lista dei figli del nodo x, definisce uno heap binomiale H' che contiene tutti i nodi contenuti nell'albero di radice x con l'eccezione del nodo x. Dato che il nodo x è eliminato da H (linea 1), allora lo heap binomiale che si ottiene come unione degli heap H e H', descritto nella figura 20.7(d), contiene tutti i nodi di H con la sola eccezione del nodo x. Infine, l'istruzione della linea 5 restituisce il nodo x.

Se lo heap H ha n nodi le istruzioni delle linee 1-4 vengono eseguite in tempo $O(\lg n)$, pertanto il tempo di esecuzione della procedura BINOMIAL-HEAP-EXTRACT-MIN risulta $O(\lg n)$.

Decremento di una chiave

La seguente procedura assegna un nuovo valore k alla chiave di un nodo x appartenente a uno heap binomiale H. La procedura segnala la presenza di una situazione di errore se il valore k è più grande del valore corrente della chiave del nodo x.

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
 1  if k > key[x]
 2      then error "la nuova chiave ha un valore maggiore della chiave corrente"
 3  key[x] ← k
 4  y ← k
 5  z ← p[y]
 6  while z ≠ NIL e key[y] < key[z]
 7      do    scambia key[y] ↔ key[z]
 8            ▷ scambia le eventuali informazioni addizionali contenute in y e z
 9            y ← z
10            z ← p[y]
```

Questa procedura, come è evidenziato dall'esempio della figura 20.8, decrementa il valore di una chiave con una tecnica analoga a quella usata nel caso degli heap binari: si fa "venire a galla" la chiave nello heap. Il primo passo della procedura è controllare che il valore della nuova chiave non sia maggiore del valore corrente: se non viene segnalata una situazione di errore allora si assegna la nuova chiave al nodo x. A questo punto la procedura comincia a scandire l'albero tramite il puntatore y che inizialmente punta al nodo x. A ogni iterazione del ciclo **while** (linee 6-10) si esaminano i valori di $key[y]$ e della chiave del padre z di y. Se y è la radice o $key[y] \geq key[z]$ allora l'albero binomiale rispetta l'ordinamento parziale dello heap. Altrimenti il nodo y non rispetta l'ordinamento parziale dello heap e pertanto la sua chiave è scambiata con la chiave del nodo padre z, assieme a tutte le informazioni addizionali associate ai due nodi. Infine la procedura assegna z a y avanzando, quindi, di un livello nell'albero, e poi comincia a eseguire le istruzioni dell'iterazione successiva.

La procedura BINOMIAL-HEAP-DECREASE-KEY richiede tempo $O(\lg n)$. Per la proprietà 2 del lemma 20.1 la profondità massima del nodo x è $\lfloor \lg n \rfloor$ quindi le istruzioni del ciclo **while** (linee 6-10) vengono ripetute al massimo per un numero di volte pari a $\lfloor \lg n \rfloor$.

Eliminazione di una chiave

L'operazione che elimina da uno heap binomiale H la chiave di un nodo x e tutta l'informazione addizionale associata è una operazione che richiede un tempo $O(\lg n)$. La seguente procedura assume che lo heap binomiale non contenga chiavi il cui valore sia uguale a $-\infty$.

```
BINOMIAL-HEAP-DELETE(H, x)
 1  BINOMIAL-HEAP-DECREASE-KEY(H, x, −∞)
 2  BINOMIAL-HEAP-EXTRACT-MIN(H)
```

Figura 20.8 *Il risultato dell'esecuzione della procedura* BINOMIAL-HEAP-DECREASE-KEY. (a) *La situazione che si presenta prima della linea 5 nella prima iterazione del ciclo* **while**. *La chiave del nodo y è stata decrementata; il suo valore (7) è minore della chiave del padre z.* (b) *Le chiavi dei due nodi sono scambiate e questa è la situazione che si presenta nella linea 5 prima della seconda iterazione. I puntatori y e z sono stati spostati di un livello nell'albero ma la proprietà dell'ordinamento heap non è ancora soddisfatta.* (c) *Dopo un ulteriore scambio ed un ulteriore spostamento dei puntatori y e z la proprietà dell'ordinamento heap è soddisfatta e pertanto il ciclo* **while** *termina.*

La procedura BINOMIAL-HEAP-DELETE prima di tutto fa in modo che la chiave associata al nodo x abbia un valore più piccolo di tutte le altre chiavi presenti nello heap. Per fare questo è sufficiente assegnare a questa chiave il valore $-\infty$. (L'Esercizio 20.2-6 affronta il caso in cui non sia possibile utilizzare il valore $-\infty$ neppure temporaneamente.) Successivamente, mediante una chiamata della procedura BINOMIAL-HEAP-DECREASE-KEY, si fa venire a galla la chiave e tutta l'informazione addizionale associata al nodo fino a che non si raggiunge una radice. Questa radice viene poi eliminata dallo heap con una chiamata della procedura BINOMIAL-HEAP-EXTRACT-MIN.

La procedura BINOMIAL-HEAP-DELETE richiede tempo $O(\lg n)$.

Esercizi

20.2-1 Si dia un esempio di due heap *binari*, con n elementi ciascuno, tali che la procedura BUILD-HEAP, richiede tempo $\Theta(n)$ per concatenare gli array dei due heap.

20.2-2 Si scrivano le istruzioni della procedura BINOMIAL-HEAP-MERGE.

20.2-3 Si mostri lo heap binomiale che si ottiene come risultato dell'operazione che inserisce la chiave di valore 24 nello heap della figura 20.7(d).

20.2-4 Si mostri lo heap binomiale che si ottiene come risultato dell'operazione che elimina la chiave di valore 28 dallo heap della figura 20.8(c).

20.2-5 Si spieghi il motivo per cui la procedura BINOMIAL-HEAP-MINIMUM non opera correttamente in presenza di chiavi con valore uguale a ∞. Si riscrivano le istruzioni della procedura in modo che possa operare correttamente anche in presenza di chiavi con valore uguale a ∞.

20.2-6 Supponiamo che non sia possibile rappresentare in alcun modo la chiave con valore uguale a $-\infty$. Si riscriva la procedura BINOMIAL-HEAP-DELETE in modo tale che possa operare correttamente anche in questo caso. La procedura deve essere eseguita ancora in tempo $O(\lg n)$.

20.2-7 Si analizzino le relazioni esistenti tra l'operazione che inserisce una chiave in uno heap binomiale e l'operazione che incrementa un numero binario. Si analizzino le relazioni esistenti tra l'operazione di unione di due heap binomiali e l'operazione di somma di due numeri binari.

20.2-8 Alla luce della risposta dell'Esercizio 20.2-7 si riscriva la procedura BINOMIAL-HEAP-INSERT in modo da non richiedere l'uso della procedura BINOMIAL-HEAP-UNION.

20.2-9 Si mostri che, se le liste delle radici sono ordinate per grado decrescente (invece che per grado crescente), tutte le operazioni definite su heap binomiali possono essere realizzate senza alcuna modifica del tempo di esecuzione asintotico.

20.2-10 Si determinino valori di ingresso per i quali le procedure BINOMIAL-HEAP-EXTRACT-MIN, BINOMIAL-HEAP-DECREASE-KEY e BINOMIAL-HEAP-DELETE sono eseguite in tempo $\Omega(\lg n)$. Si spieghi il motivo per cui il tempo di esecuzione nel caso pessimo di BINOMIAL-HEAP-INSERT, BINOMIAL-HEAP-MINIMUM e BINOMIAL-HEAP-UNION è $\Omega(\lg n)$ ma non $\Omega(\lg n)$. (Cfr. Problema 2-5.)

Problemi

20-1 ***Heap 2-3-4***

Nel Capitolo 19 sono stati introdotti gli alberi 2-3-4 in cui ogni nodo interno (con la sola eccezione della radice) ha due, tre o quattro figli e tutte le foglie sono alla stessa profondità. In questo problema ci poniamo l'obiettivo di realizzare gli ***heap 2-3-4*** come heap aggregabili.

Gli heap 2-3-4 differiscono dagli alberi 2-3-4 per il seguente motivo. Negli heap 2-3-4 le chiavi sono contenute solamente nelle foglie, inoltre ogni foglia x contiene esattamente una chiave nel campo $key[x]$. Le chiavi delle foglie non hanno un ordine particolare. Ogni nodo interno x contiene un valore $small[x]$ che è uguale al valore della chiave più piccola del sottoalbero di radice x. La radice r contiene un campo $height[r]$ in cui viene memorizzata l'altezza dell'albero. Infine, gli heap 2-3-4 sono pensati per essere mantenuti in memoria principale, pertanto non sono necessarie operazioni di lettura e scrittura del disco.

Si realizzino le operazioni seguenti (a)-(f) su heap 2-3-4 in modo tale che il tempo di esecuzione di tutte le operazioni delle parti (a)-(e) su di uno heap 2-3-4 con n elementi sia $O(\lg n)$. L'operazione UNION della parte (f) deve essere eseguita in tempo $O(\lg n)$, dove n è il numero complessivo degli elementi dei due heap.

a. MINIMUM: restituisce il puntatore alla foglia con chiave minima.

b. DECREASE-KEY: decrementa il valore di una chiave di una determinata foglia x assegnandole un valore k tale che $k \leq key[x]$.

c. INSERT: inserisce una foglia x con una chiave k.

d. DELETE elimina una foglia x.

e. EXTRACT-MIN: estrae la foglia con la chiave più piccola.

f. UNION: restituisce lo heap 2-3-4 che si ottiene unendo due heap 2-3-4. La procedura distrugge i due heap in ingresso.

20-2 *Algoritmo per il minimo albero di copertura con gli heap aggregabili*

Nel Capitolo 24 vengono presentati due algoritmi che risolvono il problema di trovare il minimo albero di copertura di un grafo non orientato. In questo problema vogliamo far vedere come gli heap aggregabili possano essere utilizzati per definire un algoritmo differente per la soluzione del problema del minimo albero di copertura.

Consideriamo un grafo non orientato $G = (V, E)$ a cui è associata una funzione peso w: $E \rightarrow \mathbb{R}$. Diciamo che $w(u, v)$ è il peso dell'arco (u, v). Si vuole trovare l'albero di copertura minimo per il grafo G: un sottoinsieme di archi $T \subseteq E$, senza cicli, che connette tutti i vertici di V e tale che il suo peso totale

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

sia minimo.

Utilizzando le tecniche introdotte nel paragrafo 24.1 si può dimostrare che la seguente procedura costruisce il minimo albero di copertura T. La procedura definisce una partizione $\{V_i\}$ dei vertici di V e, per ogni insieme di vertici V_i, considera l'insieme

$$E_i \subseteq \{(u, v) : u \in V_i \text{ oppure } v \in V_i\}$$

degli archi che incidono sui vertici che appartengono a V_i.


```
MST-MERGEABLE-HEAP(G)
T ← ∅
for tutti i vertici v_i ∈ V[G]
      do  V_i ← {v_i}
          E_i ← {(v_i, v) ∈ E[G]}
while il numero degli insiemi V_i è maggiore di 1
      do  seleziona un generico insieme V_i
          estrai l'arco (u, v) di E_i con il peso minimo
          si assuma che u ∈ V_i e v ∈ V_j
          if i ≠ j
                then  T ← T ∪ {(u, v)}
                      V_i ← V_i ∪ V_j e cancella V_j
                      E_i ← E_i ∪ E_j
```

Si descriva una realizzazione di questo algoritmo che fa uso delle operazioni degli heap aggregabili descritte nella figura 20.1. Si determini il tempo di esecuzione della soluzione proposta nell'ipotesi che gli heap aggregabili siano realizzati con heap binomiali.

Note al capitolo

Gli heap binomiali sono stati introdotti nel 1978 da Vuillemin [196]. Le proprietà degli heap binomiali sono state studiate da Brown [36, 37].

Gli heap di Fibonacci

21

Nel Capitolo 20 abbiamo visto come gli heap binomiali forniscano in tempo $O(\lg n)$ nel caso pessimo le operazioni di fusione di heap quali INSERT, MINIMUM, EXTRACT-MIN e UNION, oltre alle operazioni DECREASE-KEY e DELETE. In questo capitolo esamineremo gli heap di Fibonacci, i quali offrono le stesse operazioni ma hanno il vantaggio che quelle che non richiedono l'eliminazione di un elemento vengono effettuate in tempo ammortizzato $O(1)$.

Da un punto di vista teorico, gli heap di Fibonacci risultano particolarmente interessanti quando il numero di operazioni EXTRACT-MIN e DELETE è piccolo in confronto al numero di altre operazioni richieste. Questa situazione si verifica in molte applicazioni: per esempio, alcuni algoritmi per problemi su grafi possono chiamare DECREASE-KEY una volta per ciascun lato. Per grafi densi, che possono avere molti lati, il tempo ammortizzato $O(1)$ per ogni chiamata di DECREASE-KEY rappresenta un grosso miglioramento rispetto al tempo $\Theta(\lg n)$ nel caso pessimo degli heap binari e binomiali. Gli algoritmi asintoticamente più veloci finora conosciuti per problemi quali trovare l'albero di copertura minimo (Capitolo 24) o i cammini minimi con sorgente singola (Capitolo 25) usano in maniera essenziale gli heap di Fibonacci.

Da un punto di vista pratico, però, le costanti moltiplicative e la complessità della programmazione con gli heap di Fibonacci fanno sì che siano meno utilizzati degli ordinari heap binari (o k-ari) nella maggior parte delle applicazioni. Quindi gli heap di Fibonacci sono interessanti soprattutto da un punto di vista teorico: un alto interesse pratico lo avrebbe una struttura di dati più semplice che riuscisse a ottenere gli stessi limiti di tempo ammortizzato che si riescono a ottenere con gli heap di Fibonacci.

Uno heap di Fibonacci, analogamente a uno heap binomiale, è un insieme di alberi. In effetti, gli heap di Fibonacci sono basati sugli heap binomiali, anche se questa affermazione non è completamente esatta. Se né DECREASE-KEY né DELETE vengono chiamate in uno heap di Fibonacci, ogni albero dello heap si comporta come un albero binomiale. Gli heap di Fibonacci differiscono comunque dagli heap binomiali perché hanno una struttura con meno vincoli, permettendo dei limiti di tempo asintoticamente migliori. Il lavoro per mantenere la struttura può essere rimandato fino a quando non è conveniente eseguirlo.

Come le tabelle dinamiche del paragrafo 18.4, gli heap di Fibonacci sono un valido esempio di una struttura di dati costruita pensando alle analisi in tempo ammortizzato: l'intuizione e l'analisi delle operazioni sugli heap di Fibonacci che si trovano nel seguito del capitolo si basano moltissimo sul metodo del potenziale descritto nel paragrafo 18.3.

L'esposizione di questo capitolo presuppone la lettura del Capitolo 20 sugli heap binomiali: le specifiche per le operazioni appaiono in quel capitolo, così come la figura 20.1, che riassume i limiti di tempo delle operazioni sugli heap binari, binomiali e di Fibonacci. La nostra presentazione della struttura degli heap di Fibonacci si basa su quella della struttura

degli heap binomiali: si noterà anche che alcune delle operazioni eseguite sugli heap di Fibonacci sono analoghe a quelle eseguite sugli heap binomiali.

In maniera analoga a quanto accadeva per gli heap binomiali, anche quelli di Fibonacci non sono progettati per realizzare in maniera efficiente l'operazione SEARCH; dunque operazioni che si riferiscono a un dato nodo richiedono che venga fornito un puntatore a quel nodo come parte dell'input.

Il paragrafo 21.1 definisce gli heap di Fibonacci, ne descrive la rappresentazione e presenta la funzione potenziale utilizzata per l'analisi ammortizzata. Il paragrafo 21.2 fa vedere come realizzare le operazioni di fusione di heap e ottenere i limiti in tempo ammortizzato mostrati nella figura 20.1. Nel paragrafo 21.3 vengono presentate le due rimanenti operazioni, DECREASE-KEY e DELETE. Infine, il paragrafo 21.4 completa una parte fondamentale dell'analisi.

21.1 La struttura degli heap di Fibonacci

Analogamente a uno heap binomiale, uno ***heap di Fibonacci*** è un insieme di alberi con l'ordinamento parziale dello heap, anche se tali alberi non devono necessariamente essere binomiali. La figura 21.1(a) mostra uno heap di Fibonacci.

Diversamente da quanto avviene con gli heap binomiali, che sono alberi ordinati, gli alberi negli heap di Fibonacci hanno una radice ma non sono ordinati. Come si vede dalla figura 21.1(b), ogni nodo x contiene un puntatore $p[x]$ al padre e un puntatore $child[x]$ a uno dei suoi figli. I figli di x sono collegati da una lista circolare a doppio collegamento, che indicheremo come la ***lista dei figli*** di x. Ogni figlio y in una lista dei figli ha dei puntatori $left[y]$ e $right[y]$ che puntano, rispettivamente, ai suoi fratelli sinistro e destro. Se il nodo y è l'unico figlio, allora $left[y] = right[y] = y$; l'ordine in cui compaiono i fratelli in una lista dei figli è arbitrario.

Le liste circolari a doppio collegamento (si veda il paragrafo 11.2) presentano due vantaggi quando sono usate per gli heap di Fibonacci. Anzitutto si può togliere un nodo da una lista in tempo $O(1)$. Inoltre, date due liste siffatte, queste possono essere concatenate (o "appiccicate" insieme) in un'unica lista in tempo $O(1)$. Nella descrizione delle operazioni sugli heap di Fibonacci, ci riferiremo a tali operazioni in maniera informale, lasciando al lettore il compito di curare i dettagli della loro relazione.

Ci serviranno altri due campi per ogni nodo: il numero di figli nella lista dei figli del nodo x verrà mantenuto nel campo $degree[x]$; il campo a valori booleani $mark[x]$ indica se un nodo x ha perso un figlio dall'ultima volta in cui è diventato a sua volta figlio di un altro nodo. Non ci preoccuperemo dei dettagli su come marcare i nodi fino al paragrafo 21.3. I nodi vengono creati non marcati e un nodo x diventa non marcato ogni qualvolta diventa figlio di un altro nodo.

Si può accedere a un dato albero di Fibonacci H mediante un puntatore $min[H]$ alla radice dell'albero che contiene una chiave minima; questo nodo è chiamato il ***nodo minimo*** dello heap di Fibonacci. Se lo heap di Fibonacci è vuoto, allora $min[H] = \text{NIL}$.

Le radici di tutti gli alberi di uno heap di Fibonacci sono collegate mediante i loro puntatori *left* e *right* in una lista circolare a doppio collegamento indicata come la ***lista delle radici*** dello heap di Fibonacci. Quindi il puntatore $min[H]$ punta al nodo della lista delle radici il cui valore è minimo. L'ordine degli alberi in una lista delle radici è arbitrario.

Figura 21.1 (a) *Uno heap di Fibonacci formato da cinque alberi con ordinamento heap e 14 nodi. La linea tratteggiata indica la lista delle radici. Il nodo minimo dello heap è quello contenente la chiave 3. I tre nodi marcati sono quelli in nero. Il potenziale di questo specifico heap di Fibonacci è* $5 + 2 \cdot 3 = 11$. (b) *Una rappresentazione più completa, che mostra i puntatori p (frecce verso l'alto), child (frecce verso il basso) e left e right (frecce laterali). Questi dettagli verranno omessi nelle figure seguenti del capitolo, dato che queste informazioni possono essere tutte ricavate da quelle che appaiono in* (a).

Sfrutteremo anche un altro attributo di un heap di Fibonacci H: il numero dei nodi attualmente in H è contenuto in $n[H]$.

Funzione potenziale

Come detto prima, utilizzeremo il metodo del potenziale descritto nel paragrafo 18.3 per analizzare le prestazioni delle operazioni sugli heap di Fibonacci. Dato uno heap di Fibonacci H, indicheremo con $t(H)$ il numero di alberi nella lista delle radici di H e con $m(H)$ il numero dei nodi marcati in H. Il potenziale degli heap di Fibonacci è definito da

$$\Phi(H) = t(H) + 2\,m(H)\,. \tag{21.1}$$

Per esempio, il potenziale dello heap di Fibonacci in figura 21.1 è $5 + 2 \cdot 3 = 11$. Il potenziale di un insieme di heap di Fibonacci è dato dalla somma dei potenziali degli heap che lo costituiscono. Assumeremo che una unità di potenziale indichi una quantità costante di lavoro, e che questa costante sia grande a sufficienza per coprire il costo di tutte le varie operazioni a costo costante di cui potremmo aver bisogno.

Assumeremo poi che un'applicazione che fa uso di uno heap di Fibonacci inizi con uno heap vuoto. Il potenziale iniziale è dunque 0 e, per l'equazione (21.1), rimarrà non negativo anche nel seguito. L'equazione (18.2) garantisce che un limite superiore al costo totale ammortizzato è anche un limite superiore al costo totale reale per quella sequenza di operazioni.

Grado massimo

Nelle analisi ammortizzate riportate nei paragrafi seguenti di questo capitolo si assume che esista un limite superiore noto $D(n)$ sul grado massimo di un qualunque nodo in uno heap di Fibonacci con n nodi. L'Esercizio 21.2-3 fa vedere come, quando ci restringiamo alle sole operazioni che fondono heap, allora $D(n) = \lfloor \lg n \rfloor$. Nel paragrafo 21.3 dimostreremo che, quando sono fornite anche DECREASE-KEY e DELETE, allora $D(n) = O(\lg n)$.

21.2 Operazioni che fondono heap

In questo paragrafo descriviamo e analizziamo come sono realizzate le operazioni che fondono heap negli heap di Fibonacci. Se devono essere fornite solo le operazioni MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN e UNION, allora uno heap di Fibonacci non è altro che un insieme di alberi binomiali "non ordinati". Un ***albero binomiale non ordinato*** è simile a un albero binomiale e come quello è definito ricorsivamente. Un albero binomiale non ordinato U_0 consiste di un nodo singolo, mentre un albero binomiale non ordinato U_k consiste di due alberi binomiali non ordinati U_{k-1}, per i quali la radice di uno diventa uno *qualunque* dei figli della radice dell'altro. Le proprietà degli alberi binomiali descritte nel lemma 20.1 valgono anche nel caso non ordinato, con la seguente variazione della proprietà 4 (cfr. Esercizio 21.2-2):

4'. Dato un albero binomiale non ordinato U_k, la sua radice ha grado k, maggiore di quello di ogni altro nodo. I figli della radice sono radici di sottoalberi $U_0, U_1, \ldots, U_{k-1}$ in un ordine qualsiasi.

Quindi, se uno heap di Fibonacci con n nodi è un insieme di alberi binomiali non ordinati, allora $D(n) = \lg n$.

L'idea di fondo nelle operazioni di fusione di heap negli heap di Fibonacci è di rimandare il lavoro il più possibile. C'è un bilanciamento riguardo all'efficienza fra le realizzazioni delle varie operazioni. Se il numero di alberi in uno heap di Fibonacci è piccolo, allora possiamo rapidamente determinare il nuovo nodo minimo nel corso di un'operazione EXTRACT-MIN. Però, come abbiamo visto con gli heap binomiali nell'Esercizio 20.2-10, paghiamo un prezzo per assicurarci che il numero di alberi sia piccolo: può essere necessario un tempo $\Omega(\lg n)$ per inserire un nodo in uno heap binomiale o per unirne due. Come vedremo, non cercheremo di ristrutturare gli alberi in uno heap di Fibonacci quando inseriamo un nuovo nodo o fondiamo due heap. Rimandiamo la ristrutturazione fino all'occorrenza di un'operazione di EXTRACT-MIN, che è il momento in cui abbiamo davvero bisogno di trovare il nuovo nodo minimo.

Creazione di un nuovo heap di Fibonacci

Per realizzare uno heap di Fibonacci vuoto, la procedura MAKE-FIB-HEAP alloca e restituisce l'oggetto heap di Fibonacci H, per il quale $n[H] = 0$ e $min[H] = \text{NIL}$: non ci sono alberi in H. Dato che $t(H) = 0$ e $m(H) = 0$, il potenziale dello heap di Fibonacci vuoto è $\Phi(H) = 0$. Il costo ammortizzato di MAKE-FIB-HEAP è quindi uguale al costo reale $O(1)$.

Figura 21.2 *Inserimento di un nodo in uno heap di Fibonacci.* (a) *Uno heap di Fibonacci H.* (b) *Lo heap di Fibonacci H dopo che è stato inserito il nodo con chiave 21. Il nodo diventa l'unico componente di un albero con ordinamento heap ed è poi aggiunto alla lista delle radici, diventando il fratello sinistro della radice.*

Inserzione di un nodo

La seguente procedura inserisce un nodo x nello heap di Fibonacci H, assumendo ovviamente che il nodo sia già stato allocato e che $key[x]$ sia già stato riempito.

```
FIB-HEAP-INSERT(H, x)
 1  degree[x] ← 0
 2  p[x] ← NIL
 3  child[x] ← NIL
 4  left[x] ← x
 5  right[x] ← x
 6  mark[x] ← FALSE
 7  concatena la lista delle radici contenente x con la lista delle radici di H
 8  if min[H] = NIL oppure key[x] < key[min[H]]
 9        then min [H] ← x
10  n[H] ← n[H] + 1
```

Dopo che le linee 1-6 hanno inizializzato i campi strutturali del nodo x, rendendolo l'unico elemento di una lista circolare a doppio collegamento, la linea 7 aggiunge x alla lista delle radici di H in tempo effettivo $O(1)$. In tal modo il nodo x diventa un albero heap composto di un singolo nodo, e anche un albero binomiale non ordinato, nello heap di Fibonacci. Le linee 8-9 quindi aggiornano il puntatore al nodo minimo dello heap di Fibonacci H, se necessario. Infine, la linea 10 aumenta $n[H]$ per riflettere l'aggiunta di un nuovo nodo. La figura 21.2 mostra un nodo con chiave 21 inserito nello heap di Fibonacci di figura 21.1.

A differenza della procedura BINOMIAL-HEAP-INSERT, la FIB-HEAP-INSERT non fa nessun tentativo di ristrutturare gli alberi all'interno dello heap di Fibonacci. Se avvengono k operazioni FIB-HEAP-INSERT di seguito, vengono aggiunti alla lista delle radici k alberi di un unico nodo.

Per determinare il costo ammortizzato di FIB-HEAP-INSERT, sia H lo heap di Fibonacci di input e H' lo heap di Fibonacci risultante. Allora $t(H') = t(H) + 1$ e $m(H') = m(H)$ e l'aumento del potenziale è dato da

$$((t(H) + 1) + 2\,m(H)) - (t(H) + 2\,m(H)) = 1\ .$$

Dato che il costo reale è $O(1)$, il costo ammortizzato è $O(1) + 1 = O(1)$.

Ricerca del nodo minimo

Il nodo minimo in uno heap di Fibonacci H è indicato dal puntatore $min[H]$, cosicché può essere trovato in tempo reale $O(1)$. Dato che il potenziale di H non cambia, il costo ammortizzato di questa operazione equivale al suo costo reale $O(1)$.

Unione di due heap di Fibonacci

La seguente procedura unisce due heap di Fibonacci H_1 e H_2, distruggendoli subito dopo.

```
FIB-HEAP-UNION(H1, H2)
1   H ← MAKE-FIB-HEAP()
2   min[H] ← min[H1]
3   concatena la lista delle radici di H2 con la lista delle radici di H
4   if (min[H1] = NIL) oppure (min[H2] ≠ NIL e min[H2] < min[H1])
5         then min[H] ← min[H2]
6   n[H] ← n[H1] + n[H2]
7   rilascia gli oggetti H1 e H2
8   return H
```

Le linee 1 e 3 concatenano le liste delle radici di H_1 e H_2 creando una nuova lista delle radici H. Le linee 2, 4 e 5 creano il nodo minimo per H e la linea 6 assegna a $n[H]$ il numero totale di nodi. Gli oggetti heap di Fibonacci H_1 e H_2 sono rilasciati nella linea 7, mentre la linea 8 restituisce lo heap di Fibonacci H risultante. Come accadeva nella procedura FIB-HEAP-INSERT, non occorre ristrutturare gli alberi.

La variazione del potenziale è

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= (t(H) + 2\,m(H)) - ((t(H_1) + 2\,m(H_1)) + (t(H_2) + 2\,m(H_2))) \\ &= 0\ ,\end{aligned}$$

dato che $t(H) = t(H_1) + t(H_2)$ e $m(H) = m(H_1) + m(H_2)$. Il costo ammortizzato di FIB-HEAP-UNION è dunque uguale al suo costo reale $O(1)$.

Estrazione del nodo minimo

Il processo di estrazione del nodo minimo è la più complicata tra le operazioni presentate in questo paragrafo; è anche quella in cui il lavoro, fino ad allora rimandato, di ristrutturazione degli alberi nella lista delle radici viene infine eseguito. La procedura seguente estrae il nodo minimo; si assume per convenienza che quando un nodo è rimosso da una lista concatenata i puntatori che rimangono nella lista siano aggiornati mentre quelli nel nodo estratto restano gli stessi. Viene inoltre chiamata la procedura ausiliaria CONSOLIDATE, che sarà presentata subito dopo.


```
FIB-HEAP-EXTRACT-MIN(H)
1   z ← min[H]
2   if z ≠ NIL
3      then for ogni figlio x di z
4               do aggiungi x alla lista delle radici di H
5                  p[x] ← NIL
6               togli z dalla lista delle radici di H
7               if z = right[z]
8                  then min[H] ← NIL
9                  else min[H] ← right[z]
10                       CONSOLIDATE(H)
11              n[H] ← n[H] − 1
12  return z
```

Come mostra la figura 21.3, FIB-HEAP-EXTRACT-MIN trasforma in una radice tutti i figli del nodo minimo, ed elimina poi il nodo minimo dalla lista delle radici. In seguito, ristruttura la lista delle radici collegando radici con lo stesso grado fino a quando rimane al più una radice per ogni grado.

Cominciamo nella linea 1 salvando un puntatore z al nodo minimo: questo puntatore è poi restituito alla fine. Se z = NIL, allora lo heap di Fibonacci H è già vuoto e abbiamo finito; altrimenti, come nel caso della procedura BINOMIAL-HEAP-EXCTRACT-MIN, si cancella il nodo z da H rendendo tutti i figli di z radici di H nelle linee 3-5 (mettendoli nella lista delle radici) e togliendo z dalla lista delle radici nella linea 6. Se $z = right[z]$ dopo la linea 6, allora z era l'unico nodo nella lista delle radici e non aveva figli, così tutto quello che rimane da fare è svuotare lo heap di Fibonacci nella linea 8 prima di restituire z. Altrimenti, assegniamo al puntatore $min[H]$ nella lista delle radici un nodo diverso da z (in questo caso, $right[z]$). La figura 21.3(b) mostra lo heap di Fibonacci della figura 21.3(a) dopo che è stata eseguita la linea 9.

Il passo successivo, in cui si riduce il numero di alberi nello heap di Fibonacci, consiste nel ristrutturare la lista delle radici di H, cosa che avviene mediante la chiamata di CONSOLIDATE(H). La ristrutturazione della lista delle radici consiste nell'eseguire ripetutamente i passi seguenti fino a quando ogni radice nella lista delle radici ha grado diverso.

1. Trova due radici x e y nella lista delle radici con lo stesso grado, dove $key[x] \le key[y]$.
2. Collega y a x: togli y dalla lista delle radici e fallo diventare un figlio di x. Questa operazione è eseguita dalla procedura FIB-HEAP-LINK. Il campo $degree[x]$ è incrementato, mentre viene tolta la marcatura da y, se presente.

La procedura CONSOLIDATE usa un array ausiliario $A[0\,..\,D(n[H])]$; se $A[i] = y$, allora y è una radice con $degree[y] = i$.

Figura 21.3 *L'azione di* FIB-HEAP-EXCTRACT-MIN. (**a**) *Uno heap di Fibonacci H.* (**b**) *La situazione dopo che il nodo minimo z è stato tolto dalla lista delle radici e i suoi figli sono stati aggiunti alla lista delle radici.* (**c**)-(**e**) *L'array A e gli alberi dopo ognuna delle prime tre iterazioni del ciclo* **for** *delle righe* 3-13 *della procedura* CONSOLIDATE. *La lista delle radici viene analizzata a partire dal nodo minimo e seguendo i puntatori right. Ogni immagine mostra i valori di w e x alla fine di un'iterazione.* (**f**)-(**h**) *L'iterazione successiva del ciclo* **for**, *con i valori di w e x alla fine di ogni iterazione del ciclo* **while** *delle linee* 6-12. *L'immagine* (**f**) *mostra la situazione dopo il primo passaggio del ciclo* **while**. *Il nodo con la chiave* 23 *è stato collegato al nodo con chiave* 7, *che adesso è puntato da x. Nell'immagine* (**g**) *il nodo con chiave* 17 *è stato collegato al nodo con chiave* 7, *a cui ancora punta x. Nell'immagine* (**h**) *il nodo con chiave* 24 *è stato collegato al nodo con chiave* 7. *Dato che fino a quel momento* $A[3]$ *non puntava a nessun nodo, alla fine del ciclo* **for** *viene fatto puntare alla radice dell'albero risultante.* (**i**)-(**l**) *La situazione dopo ognuna delle quattro iterazioni del ciclo* **while**. (**m**) *Lo heap di Fibonacci H dopo la ricostruzione della lista delle radici a partire dall'array A e l'individuazione del nuovo puntatore a* $min[H]$.

CONSOLIDATE(H)

```
1  for i ← 0 to D(n[H])
2      do A[i] ← NIL
3  for ogni nodo w nella lista delle radici di H
4      do x ← w
5          d ← degree[x]
6          while A[d] ≠ NIL
```

Figura 21.3 *(continua)*

```
7              do y ← A[d]
8                  if key[x] > key[y]
9                      then scambia x ↔ y
10                     FIB-HEAP-LINK(H, y, x)
11                     A[d] ← NIL
12                     d ← d + 1
13         A[d] ← x
14 min[H] ← NIL
15 for i ← 0 to D(n[H])
16     do if A[i] ≠ NIL
17         then aggiungi A[i] alla lista delle radici di H
18              if min[H] = NIL oppure key[A[i]] < key[min[H]]
19                  then min[H] ← A[i]
```

FIB-HEAP-LINK(H, y, x)

```
1  togli y dalla lista delle radici di H
2  rendi y un figlio di x, incrementando degree[x]
3  mark[y] ← FALSE
```

Scendendo nel dettaglio, la procedura CONSOLIDATE opera nel seguente modo: nelle linee 1-2 inizializza A mettendo ogni elemento a NIL; dopo l'analisi di una qualunque radice w, otteniamo un albero avente per radice un qualche nodo x, che può coincidere o no con w, e l'elemento $A[degree[x]]$ dell'array viene fatto puntare a x. Nel ciclo **for** delle linee 3-13, si esamina ogni radice w nella lista delle radici. L'invariante mantenuto nel corso di ogni iterazione del ciclo **for** è che il nodo x è la radice dell'albero che contiene il nodo w. Il ciclo **while** delle linee 6-12 preserva l'invariante $d = degree[x]$ (tranne che nella linea 11, come vedremo tra un attimo). In ogni iterazione del ciclo **while**, $A[d]$ punta a una radice y. Dato che $d = degree[x] = degree[y]$, vogliamo collegare x e y. Il risultato dell'operazione di collegamento è che chi ha la chiave più piccola tra x e y diventa il padre dell'altro, e quindi le linee 8-9 scambiano i puntatori a x e y se necessario. Nel seguito, y è collegato a x dalla chiamata a FIB-HEAP-LINK(H, x, y) nella linea 10, che incrementa $degree[x]$ ma lascia $degree[y]$ a d. Dato che il nodo y non è più una radice, il puntatore a esso nell'array A è tolto nella linea 11. Dato che il valore di $degree[x]$ è incrementato dalla chiamata di FIB-HEAP-LINK, la linea 12 ricrea l'invariante $d = degree[x]$. Si ripete il ciclo **while** fino a quando $A[d]$ = NIL, nel qual caso non esiste nessun'altra radice con lo stesso grado di x. Nella linea 13 assegniamo x a $A[d]$ ed eseguiamo la successiva iterazione del ciclo **for**. Le figure 21.3(c)-(e) mostrano l'array H e gli alberi risultanti dopo le prime tre iterazioni del ciclo **for** delle linee 3-13. Nella successiva iterazione del ciclo **for** avvengono tre collegamenti; il loro risultato è mostrato nelle figure 21.3 (f)-(h). Le figure 21.3 (i)-(l) mostrano il risultato delle quattro successive iterazioni del ciclo **for**.

A questo punto rimane soltanto da ripulire. Dopo che il ciclo **for** delle linee 3-13 è completato, la linea 14 vuota la lista delle radici e le linee 15-19 la ricostruiscono. Lo heap di Fibonacci risultante è mostrato nella figura 21.3(m). Dopo aver ristrutturato la lista delle radici, FIB-HEAP-EXTRACT-MIN termina decrementando $n[H]$ nella linea 11 e restituendo nella linea 12 un puntatore al nodo cancellato z.

Va notato che se tutti gli alberi nello heap di Fibonacci erano alberi binomiali non ordinati prima che la FIB-HEAP-EXTRACT-MIN fosse eseguita, allora questi rimangono tali anche dopo. La struttura degli alberi può essere cambiata solo in due modi. Anzitutto, nelle linee 3-5 di FIB-HEAP-EXTRACT-MIN, ogni figlio x della radice z diventa una radice: per quanto affermato nell'Esercizio 21.2-2, ogni nuovo albero è anch'esso un albero binomiale non ordinato. In secondo luogo, gli alberi sono collegati da FIB-HEAP-LINK solo se hanno lo stesso grado: dato che tutti gli alberi sono alberi binomiali non ordinati prima di essere collegati, due alberi le cui radici hanno ognuna k figli devono avere la struttura di U_k. L'albero risultante quindi ha la struttura di U_{k+1}.

Possiamo adesso far vedere che il costo ammortizzato per estrarre il nodo minimo da uno heap di Fibonacci di n nodi è $O(D(n))$. Sia H lo heap di Fibonacci prima dell'esecuzione di FIB-HEAP-EXCTRACT-MIN.

Il costo reale dell'estrazione del nodo minimo può essere descritto come segue. Un contributo $O(D(n))$ viene dal fatto che in FIB-HEAP-EXTRACT-MIN vengono elaborati al più $D(n)$ figli del nodo minimo e dal lavoro svolto in CONSOLIDATE nelle linee 1-2 e 14-19. Rimane da analizzare il costo del ciclo **for** nelle linee 3-13. La dimensione della lista delle radici dopo la chiamata di CONSOLIDATE è al più $D(n) + t(H) - 1$, dato che consiste dei $t(H)$ nodi della lista originale di radici, meno il nodo radice estratto, più i figli del nodo estratto, il cui numero è al più $D(n)$. Ogni volta che si attraversa il ciclo **while** delle linee 6-12, una delle radici è collegata a un'altra, e quindi la quantità totale di lavoro eseguita nel ciclo **for** è al più proporzionale a $D(n) + t(H)$. Quindi il lavoro totale reale è $O(D(n) + t(H))$.

Il potenziale prima dell'estrazione del nodo è $t(H) + 2m(H)$, mentre il potenziale dopo è al più $(D(n) + 1) + 2m(H)$, dato che rimangono al più $D(n) + 1$ radici e nessun nodo viene marcato durante l'operazione. Il costo ammortizzato è dunque al più

$$\begin{aligned} & O(D(n) + t(H)) + ((D(n) + 1) + 2\,m(H)) - (t(H) + 2\,m(H)) \\ &\quad = O(D(n)) + O(t(H)) - t(H) \\ &\quad = O(D(n)) , \end{aligned}$$

dato che possiamo incrementare le unità di potenziale fino a farle dominare le costanti nascoste all'interno di $O(t(H))$. Intuitivamente, il costo dell'esecuzione di ogni collegamento è pagato dalla riduzione in potenziale dovuta al fatto che si diminuisce di uno il numero delle radici.

Esercizi

21.2-1 Si mostri lo heap di Fibonacci risultante dalla chiamata di FIB-HEAP-EXTRACT-MIN sullo heap di Fibonacci mostrato nella figura 21.3(m).

21.2-2 Si dimostri che il lemma 20.1 vale per alberi binomiali non ordinati, nel caso che la proprietà 4' rimpiazzi la proprietà 4.

21.2-3 Si mostri che nel caso siano fornite solo le operazioni di fusione di heap, il grado massimo $D(n)$ in uno heap di Fibonacci con n nodi è al più $\lfloor \lg n \rfloor$.

21.2-4 Il professor McGee ha pensato a una nuova struttura di dati basata sugli heap di Fibonacci. Uno heap di McGee ha la stessa struttura di uno heap di Fibonacci e offre le stesse operazioni di fusione degli heap. Le operazioni vengono realizzate in maniera analoga a quelle sugli heap di Fibonacci, tranne che l'inserzione e l'unione eseguono la ristrutturazione come loro ultimo passo. Qual è il tempo di esecuzione delle operazioni sugli heap di McGee nel caso peggiore? Quanto è nuova la struttura di dati del professore?

21.2-5 Si mostri che quando l'unica operazione su chiavi è il confronto fra due chiavi (come è il caso di tutte le realizzazioni di questo capitolo) non tutte le operazioni che fondono heap possono essere eseguite in tempo ammortizzato $O(1)$.

21.3 Decremento di una chiave ed eliminazione di un nodo

In questo paragrafo mostreremo come decrementare la chiave di un nodo in uno heap di Fibonacci in tempo ammortizzato $O(1)$ e come eliminare un nodo qualunque da uno heap di Fibonacci con n nodi in tempo ammortizzato $O(D(n))$. Queste operazioni non preservano la proprietà in base alla quale tutti gli alberi in uno heap di Fibonacci sono alberi binomiali non ordinati. Lo sono quasi, comunque, perché possiamo limitare il grado massimo $D(n)$ con $O(\lg n)$. Provare questo limite implicherà che FIB-HEAP-EXTRACT-MIN e FIB-HEAP-DELETE vengono eseguite in tempo ammortizzato $O(\lg n)$.

Decremento di una chiave

Nella procedura seguente per l'operazione FIB-HEAP-DECREASE-KEY assumiamo, come prima, che togliere un nodo da una lista concatenata non cambi nessuno dei campi strutturali nel nodo rimosso.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > key[x]
2      then error "la nuova chiave è maggiore della chiave corrente"
3  key[x] ← k
4  y ← p[x]
5  if y ≠ NIL e key[x] < key[y]
6      then CUT(H, x, y)
7           CASCADING-CUT(H, y)
8  if key[x] < key[min[H]]
9      then min[H] ← x
```

```
CUT(H, x, y)
1  togli x dalla lista dei figli di y, decrementando degree[y]
2  aggiungi x alla lista delle radici di H
3  p[x] ← NIL
4  mark[x] ← FALSE
```

```
CASCADING-CUT(H, y)
1  z ← p[y]
2  if z ≠ NIL
3      then if mark[y] = FALSE
4               then mark[y] = TRUE
5               else CUT(H, y, z)
6                    CASCADING-CUT(H, z)
```

La procedura FIB-HEAP-DECREASE-KEY compie i seguenti passi: le linee 1-3 controllano che la nuova chiave non sia maggiore della chiave corrente di x e poi assegnano la nuova chiave a x. Se x è una radice o se $key[x] \geq key[y]$, dove y è il padre di x, allora non occorrono cambi strutturali, dato che l'ordine dello heap non è stato violato. Le linee 4-5 controllano questa condizione.

Se l'ordine dello heap è stato violato, possono verificarsi molti cambiamenti: si inizia *tagliando* x nella linea 6. La procedura CUT "taglia" il legame fra x e il padre y, rendendo x una radice.

Per ottenere i limiti di tempo desiderati usiamo i campi *mark*, che aiutano a produrre il seguente effetto. Supponiamo che x sia un nodo che ha avuto la seguente storia:

1. in un certo istante, x era una radice,
2. poi x è stato collegato a un altro nodo,
3. poi due figli di x sono stati rimossi con dei tagli.

Non appena il secondo figlio viene perso, x viene tagliato dal padre, il che lo rende una nuova radice. Il campo $mark[x]$ è TRUE se si sono fatti i passi 1 e 2 e un figlio di x è stato tagliato. La procedura CUT, quindi, cancella $mark[x]$ nella linea 4, dato che esegue il passo 1. (Possiamo ora vedere perché la linea 3 di FIB-HEAP-LINK cancella $mark[y]$: il nodo y viene collegato e quindi esegue il passo 2. La volta successiva in cui un figlio di y è tagliato, $mark[y]$ verrà messo a TRUE).

Non abbiamo ancora finito, perché x potrebbe essere il secondo figlio tagliato dal padre y dal momento in cui y era stato collegato a un altro nodo. Dunque, la linea 7 di FIB-HEAP-DECREASE-KEY esegue un'operazione di ***taglio a cascata*** su y. Se y è una radice, allora il test nella linea 2 di CASCADING-CUT causa la terminazione della procedura. Se y non è marcato, la procedura lo marca nella linea 4, dato che il suo primo figlio è stato appena tagliato, e termina. Se y è marcato, invece, ha appena perso il suo secondo figlio; y è tagliato nella linea 5 e CASCADING-CUT si chiama ricorsivamente nella linea 6 su z, padre di y. La procedura CASCADING-CUT risale lungo l'albero fino a quando non trova una radice o un nodo non marcato.

Una volta che tutti i tagli a cascata sono avvenuti, le linee 8 e 9 di FIB-HEAP-DECREASE-KEY terminano aggiornando $min[H]$ se necessario.

La figura 21.4 mostra l'esecuzione di due chiamate di FIB-HEAP-DECREASE-KEY, a partire dallo heap di Fibonacci mostrato nella figura 21.4(a). La prima chiamata, mostrata nella figura 21.4(b), non richiede tagli a cascata. La seconda chiamata, mostrata nella figura 21.4(c)-(e), invoca due tagli a cascata.

Adesso dimostreremo che il costo ammortizzato di FIB-HEAP-DECREASE-KEY è soltanto $O(1)$. Cominceremo determinando il costo reale. La procedura FIB-HEAP-DECREASE-KEY impiega tempo $O(1)$, oltre al tempo di esecuzione dei tagli a cascata. Supponiamo che CASCADING-CUT sia ricorsivamente chiamata c volte a partire da una data chiamata di FIB-HEAP-DECREASE-KEY. Ogni chiamata di CASCADING-CUT impiega tempo $O(1)$ senza contare le chiamate ricorsive. Quindi, il costo reale di FIB-HEAP-DECREASE-KEY, comprese tutte le chiamate ricorsive, è $O(c)$.

Calcoliamo adesso il cambio nel potenziale. Sia H lo heap di Fibonacci prima dell'esecuzione di FIB-HEAP-DECREASE-KEY. Ogni chiamata ricorsiva di CASCADING-CUT, tranne l'ultima, taglia un nodo marcato e rimuove la marca. Dopodiché, rimangono $t(H) + c$ alberi ($t(H)$ alberi di partenza, $c - 1$ alberi prodotti dai tagli in cascata, e l'albero con radice in x) e al più $m(H) - c + 2$ nodi marcati ($c - 1$ sono stati smarcati dai tagli in cascata e l'ultima chiamata di CASCADING-CUT può aver marcato un nodo). Il cambio in potenziale è quindi al più

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2\,m(H)) = 4 - c\ .$$

Dunque il costo ammortizzato di FIB-HEAP-DECREASE-KEY è al più

$$O(c) + 4 - c = O(1)\ ,$$

dato che possiamo incrementare le unità di potenziale fino a farle dominare le costanti nascoste all'interno di $O(c)$.

Figura 21.4 *Due chiamate di* FIB-HEAP-DECREASE-KEY. (**a**) *Lo heap di Fibonacci iniziale.* (**b**) *Viene decrementata a* 15 *la chiave del nodo che aveva chiave* 46; *il nodo diventa una radice ed il suo genitore (con chiave* 24*) viene marcato.* (**c**)-(**e**) *Viene decrementata a* 5 *la chiave del nodo che aveva chiave* 35; *nell'immagine* (**c**) *il nodo, che adesso ha chiave* 5, *diventa una radice. Il suo genitore con chiave* 26 *era marcato e dunque avviene un taglio a cascata. Il nodo con chiave* 26 *viene tagliato dal suo genitore e diventa una radice non marcata in* (**d**). *Avviene un altro taglio a cascata, dal momento che anche il nodo con chiave* 24 *è marcato. Il nodo è tagliato dal suo genitore e diventa una radice nell'immagine* (**e**). *A questo punto cessano i tagli a cascata, dato che il nodo con chiave* 7 *è una radice. (Anche se questo nodo non fosse stato una radice i tagli a cascata si sarebbero fermati, dato che non è marcato.) Il risultato dell'esecuzione di* FIB-HEAP-DECREASE-KEY *è mostrato nell'immagine* (**e**), *dove min*[*H*] *punta al nuovo nodo minimo.*

Si capisce adesso come mai la funzione potenziale sia stata definita in modo da includere un termine che è il doppio del numero dei nodi marcati. Un nodo marcato y viene smarcato quando è tagliato da un taglio a cascata, cosicché il potenziale è ridotto di 2. Un'unità di potenziale paga il taglio e la smarcatura, e l'altra unità compensa l'incremento di un'unità di potenziale dovuto al fatto che il nodo y diventa una radice.

Eliminazione di un nodo

È semplice eliminare un nodo da uno heap di Fibonacci di n nodi in tempo ammortizzato $O(D(n))$, come fa la procedura seguente; assumiamo che al momento della chiamata nessuna chiave abbia valore $-\infty$ nello heap di Fibonacci.

```
FIB-HEAP-DELETE(H, x)
  1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
  2  FIB-HEAP-EXTRACT-MIN(H)
```


FIB-HEAP-DELETE è analogo a BINOMIAL-HEAP-DELETE: rende x il nodo minimo nello heap di Fibonacci assegnandogli il valore $-\infty$. Il nodo x è poi rimosso dallo heap di Fibonacci dalla procedura FIB-HEAP-EXTRACT-MIN. Il tempo ammortizzato di FIB-HEAP-DELETE è dato dalla somma del tempo ammortizzato $O(1)$ di FIB-HEAP-DECREASE-KEY e del tempo ammortizzato $O(D(n))$ di FIB-HEAP-EXTRACT-MIN.

Esercizi

21.3-1 Supponiamo che una radice x in uno heap di Fibonacci sia marcata. Si spieghi in che modo x può essere diventata una radice marcata, e se ne deduca che per l'analisi non fa differenza che x sia marcata, anche se non è una radice che era stata in precedenza collegata a un altro nodo e ha poi perso un figlio.

21.3-2 Giustificare il tempo ammortizzato $O(1)$ di FIB-HEAP-DECREASE-KEY usando il metodo degli aggregati del paragrafo 18.1.

21.4 Limitazione del grado massimo

Per dimostrare che il tempo ammortizzato di FIB-HEAP-EXTRACT-MIN e FIB-HEAP-DELETE è $O(\lg n)$, dobbiamo far vedere che il limite superiore $D(n)$ al grado di un qualunque nodo di uno heap di Fibonacci con n nodi è $O(\lg n)$. L'Esercizio 21.2-3 ci assicura che $D(n) = \lfloor \lg n \rfloor$ nel caso in cui tutti gli alberi nello heap di Fibonacci sono alberi binomiali non ordinati. I tagli che avvengono in FIB-HEAP-DECREASE-KEY, però, possono far sì che alcuni alberi all'interno dello heap di Fibonacci violino le proprietà di un albero binomiale. In questo paragrafo mostreremo che, dal momento che tagliamo un nodo dal padre non appena perde due figli, $D(n)$ è effettivamente $O(\lg n)$. Più precisamente, faremo vedere che $D(n) \le \lfloor \log_\phi n \rfloor$, dove $\phi = (1 + \sqrt{5})/2$.

L'idea di fondo per l'analisi è la seguente: per ogni nodo x all'interno di uno heap di Fibonacci, indichiamo con size(x) il numero dei nodi, x compreso, contenuti nel sottoalbero con radice in x. (Si noti che x non fa necessariamente parte della lista delle radici: può essere un qualunque nodo.) Faremo vedere che size(x) è esponenziale in *degree*[x]. Va tenuto presente che *degree*[x] contiene sempre il valore effettivo del grado di x.

Lemma 21.1

Sia x un nodo in uno heap di Fibonacci e supponiamo che *degree*[x] sia uguale a k. Indichiamo con $y_1, y_2, \ldots, y_k$ i figli di x nell'ordine con il quale sono stati collegati a x, dal meno recente al più recente. Quindi *degree*[y_1] ≥ 0 e *degree*[y_i] $\ge i - 2$ per $i = 2, 3, \ldots, k$.

Dimostrazione. Ovviamente *degree*[y_1] ≥ 0. Dato $i \ge 2$, notiamo che quando y_i è stato collegato a x, tutti i vari $y_1, y_2, \ldots, y_{i-1}$ erano figli di x, per cui dovevamo avere *degree*[x] $\ge i - 1$. Il nodo y_i è collegato a x solo se *degree*[x] = *degree*[y_i], per cui in quel momento avevamo *degree*[y_i] $\ge i - 1$. Da quel momento il nodo y_i ha perso al più un figlio, dato che se avesse perso due figli sarebbe stato tagliato da x. Ne concludiamo che *degree*[y_i] $\ge i - 2$. ■

Arriviamo infine alla parte dell'analisi che giustifica il nome di "heap di Fibonacci". Nel paragrafo 2.2 il k-esimo numero di Fibonacci è stato definito per mezzo della ricorrenza

$$F_k = \begin{cases} 0 & \text{se } k = 0 , \\ 1 & \text{se } k = 1 , \\ F_{k-1} + F_{k-2} & \text{se } k \geq 2 . \end{cases}$$

Il seguente lemma esprime in un altro modo F_k.

Lemma 21.2

Per tutti gli interi $k \geq 0$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i .$$

Dimostrazione. La dimostrazione procede per induzione su k. Per $k = 0$

$$\begin{aligned} 1 + \sum_{i=0}^{0} F_i &= 1 + F_0 \\ &= 1 + 0 \\ &= 1 \\ &= F_2 . \end{aligned}$$

Assumiamo per ipotesi induttiva che $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, da cui discende che

$$\begin{aligned} F_{k+2} &= F_k + F_{k+1} \\ &= F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) \\ &= 1 + \sum_{i=0}^{k} F_i . \end{aligned}$$

■

Il seguente lemma e il suo corollario completano l'analisi. Il lemma usa la disuguaglianza (dimostrata nell'Esercizio 2.2-8)

$$F_{k+2} \geq \phi^k ,$$

dove ϕ è il rapporto aureo definito nell'equazione (2. 14) come $\phi = (1 + \sqrt{5})/2 = 1.61803\ldots$

Lemma 21.3

Sia x un nodo in uno heap di Fibonacci e sia $k = degree[x]$. Allora $\text{size}(x) \geq F_{k+2} \geq \phi^k$ dove $\phi = (1 + \sqrt{5})/2$.

Dimostrazione. Sia s_k il valore minimo possibile di size(z) fra tutti i nodi z tali che $degree[z] = k$. Banalmente $s_0 = 1$, $s_1 = 2$ e $s_2 = 3$. Il numero s_k è al più size(x). Indichiamo come nel lemma 21.1 i figli di x con $y_1, y_2, \ldots, y_k$, nell'ordine in cui sono stati collegati a x. Per calcolare un limite

inferiore su size(x), sommiamo 1 per il nodo stesso, e 1 per il primo figlio y_1 (per il quale $\text{size}(y_1) \geq 1$) e poi applichiamo il lemma 21.1 per gli altri figli. Abbiamo così che

$$\begin{aligned} \text{size}(x) &\geq s_k \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} . \end{aligned}$$

Dimostreremo adesso per induzione su k che $s_k \geq F_{k+2}$ per tutti gli interi non negativi k. I casi di base per $k = 0$ e $k = 1$ sono ovvi. Nel passo induttivo assumiamo che $k \geq 2$ e che $s_i \geq F_{i+2}$ per $i = 0, 1, \ldots, k-1$. Abbiamo dunque che

$$\begin{aligned} s_k &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &\geq 2 + \sum_{i=2}^{k} F_i \\ &= 1 + \sum_{i=0}^{k} F_i \\ &= F_{k+2} . \end{aligned}$$

L'ultima uguaglianza discende dal lemma 21.2.

Abbiamo così dimostrato che $\text{size}(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ■

Corollario 21.4

Il grado massimo $D(n)$ di un nodo in uno heap di Fibonacci con n nodi è $O(\lg n)$.

Dimostrazione. Sia x un nodo in uno heap di Fibonacci con n nodi e sia $k = degree[x]$. Dal lemma 21.3 discende che $n \geq \text{size}(x) \geq \phi^k$. Utilizzando i logaritmi in base ϕ abbiamo che $k \leq \log_\phi n$ (in realtà, dato che k è un intero, abbiamo che $k \leq \lfloor \log_\phi n \rfloor$). Il grado massimo $D(n)$ di un qualunque nodo è dunque $O(\lg n)$. ■

Esercizi

21.4-1 Il professor Pinocchio afferma che l'altezza di uno heap di Fibonacci con n nodi è $O(\lg n)$. Si dimostri che il professore è in errore mostrando, per un qualunque intero positivo n, una sequenza di operazioni sugli heap di Fibonacci che crea uno heap di Fibonacci formato da un unico albero che è una catena lineare di n nodi.

21.4-2 Immaginiamo di generalizzare la regola dei tagli a cascata in maniera tale da tagliare un nodo x dal padre non appena perde il suo k-esimo figlio, per un dato intero costante k. (La regola nel paragrafo 21.3 utilizza $k = 2$.) Per quali valori di k si ottiene che $D(n) = O(\lg n)$?

Problemi

21-1 *Realizzazione alternativa della eliminazione*

Il professor Pisano ha proposto la seguente variante della procedura FIB-HEAP-DELETE, affermando che l'esecuzione è più veloce quando il nodo che viene cancellato non è il nodo a cui punta *min*[*H*].

```
PISANO-DELETE(H, x)
1  if x = min[H]
2      then FIB-HEAP-EXTRACT-MIN(H)
3      else y ← p[x]
4           if y ≠ NIL
5               then CUT(H, x, y)
6                    CASCADING-CUT(H, y)
7           aggiungi la lista dei figli di x alla lista delle radici di H
8           togli x dalla lista delle radici di H
```

a. L'affermazione del professore che questa procedura viene eseguita più velocemente dipende in parte dall'ipotesi che la linea 7 possa essere realizzata in tempo effettivo $O(1)$. Cosa c'è di sbagliato in questa ipotesi?

b. Si fornisca un buon limite superiore al tempo effettivo di PISANO-DELETE nel caso in cui $x \neq min[H]$. Il limite dovrebbe essere espresso in termini di *degree*[*x*] e del numero *c* di chiamate della procedura CASCADING-CUT.

c. Sia *H'* lo heap di Fibonacci ottenuto dopo l'esecuzione di PISANO-DELETE(*H*, *x*). Supponendo che *x* non sia una radice, si limiti il potenziale di *H'* esprimendolo in termini di *degree*[*x*], *c*, *t*(*H*) e *m*(*H*).

d. Si concluda che il tempo ammortizzato per PISANO-DELETE non è asintoticamente migliore di FIB-HEAP-DELETE, anche nel caso in cui $x \neq min[H]$.

21-2 *Altre operazioni sugli heap di Fibonacci*

Vorremmo fornire due nuove operazioni per gli alberi di Fibonacci senza cambiare il tempo ammortizzato di esecuzione delle altre operazioni.

a. Si realizzi in maniera efficiente l'operazione FIB-HEAP-CHANGE-KEY(*H*, *x*, *k*), che cambia la chiave del nodo *x* assegnandole il valore *k*, e se ne analizzi il tempo di esecuzione ammortizzato nei casi in cui *k* sia rispettivamente maggiore, minore o uguale a *key*[*x*].

b. Si fornisca una realizzazione efficiente per l'operazione FIB-HEAP-PRUNE(*H*, *r*), che elimina *min*(*r*, *n*[*H*]) nodi da *H*. Non dovrebbero essere fatte ipotesi su quali nodi sono cancellati. Si analizzi il tempo di esecuzione ammortizzato della realizzazione. (*Suggerimento*: potrebbe essere necessario modificare la struttura di dati e la funzione potenziale).

Note al capitolo

Gli heap di Fibonacci sono stati introdotti da Fredman e Tarjan [75]. Il loro articolo descrive inoltre l'applicazione degli heap di Fibonacci al problema dei cammini minimi con sorgente singola, dei cammini minimi fra tutte le coppie, dell'abbinamento su grafi bipartiti e pesati e dell'albero di copertura minimo.

In seguito Driscoll, Gabow, Shrairman e Tarjan [58] hanno sviluppato gli "heap rilassati" come alternativa agli heap di Fibonacci. Esistono due tipi di heap rilassati: uno ha gli stessi limiti in tempo ammortizzato degli heap di Fibonacci; l'altro permette di eseguire DECREASE-KEY in tempo $O(1)$ nel caso pessimo (non ammortizzato) e EXTRACT-MIN e DELETE in tempo $O(\lg n)$ nel caso pessimo. Gli heap rilassati hanno anche ulteriori vantaggi rispetto agli heap di Fibonacci nel caso di algoritmi paralleli.

Strutture di dati per insiemi disgiunti

22

Alcune applicazioni devono raggruppare n elementi distinti in una collezione di insiemi disgiunti. In questo contesto due operazioni sono importanti: determinare a quale insieme appartiene un dato elemento e unire due insiemi. Questo capitolo esplora i metodi per il mantenimento di una struttura di dati che fornisce queste operazioni.

Il paragrafo 22.1 descrive le operazioni offerte da una struttura di dati per insiemi disgiunti e presenta una semplice applicazione. Nel paragrafo 22.2 diamo uno sguardo a una semplice realizzazione per insiemi disgiunti effettuata mediante liste concatenate. Nel paragrafo 22.3 viene data una rappresentazione piu efficiente che usa alberi radicati: il tempo di esecuzione che si ottiene utilizzando la rappresentazione ad albero è lineare per tutti gli scopi pratici ma teoricamente è sovralineare. Il paragrafo 22.4 definisce e discute la funzione di Ackermann e la sua inversa, dalla crescita molto lenta, che compare nel tempo di esecuzione delle operazioni con la realizzazione ad alberi, e quindi usa l'analisi ammortizzata per dimostrare un limite superiore leggermente più debole sul tempo di esecuzione.

22.1 Operazioni su insiemi disgiunti

Una ***struttura di dati per insiemi disgiunti*** mantiene una collezione $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ di insiemi dinamici disgiunti. Ogni insieme è identificato da un ***rappresentante***, che è un dato membro dell'insieme. In alcune applicazioni non è importante quale membro viene utilizzato come rappresentante: imponiamo soltanto che se cerchiamo due volte il rappresentante di un insieme dinamico, senza modificare l'insieme stesso fra le due richieste, si ottenga la stessa risposta entrambe le volte. In altre applicazioni ci può essere una regola descritta in precedenza, per determinare il rappresentante, quale ad esempio quella di scegliere l'elemento più piccolo dell'insieme (assumendo ovviamente che gli elementi possano essere ordinati).

Come per le altre realizzazioni con insiemi dinamici che abbiamo studiato, ogni elemento di un insieme è rappresentato da un oggetto. Dato un oggetto x, vorremmo offrire le seguenti operazioni:

MAKE-SET(x) crea un nuovo insieme il cui unico membro (e quindi rappresentante) è x. Dal momento che gli insiemi sono disgiunti, dobbiamo richiedere che x non sia già in un insieme.

UNION(x, y) unisce due insiemi dinamici che contengono x e y, ad esempio S_1 e S_2, in un nuovo insieme che è l'unione di questi due insiemi. Si suppone che i due insiemi siano disgiunti prima dell'esecuzione dell'operazione. Il rappresentante dell'insieme

risultante è un membro di $S_x \cup S_y$, benché molte realizzazioni di UNION scelgano il rappresentante di S_x o di S_y come nuovo rappresentante. Dal momento che abbiamo richiesto che gli insiemi in una collezione siano disgiunti, l'operazione "distrugge" gli insiemi S_x e S_y, togliendoli dalla collezione $\mathcal{S}$.

FIND-SET(x) restituisce un puntatore al rappresentante dell'(unico) insieme contenente x.

In tutto il capitolo analizzeremo i tempi di esecuzioni delle strutture di dati per insiemi disgiunti in termini di due parametri: n, il numero di operazioni MAKE-SET, e m, il numero totale di operazioni MAKE-SET, UNION e FIND-SET. Dato che gli insiemi sono disgiunti, ogni operazione UNION riduce di uno il numero degli insiemi; dopo $n-1$ operazioni UNION, dunque, rimane un unico insieme. Il numero di operazioni UNION è perciò al più $n-1$. Si faccia inoltre attenzione che, dal momento che le operazioni MAKE-SET sono incluse nel numero totale di m operazioni, ne discende che $m \geq n$.

Un'applicazione delle strutture per insiemi disgiunti

Una delle molte applicazioni delle strutture di dati per insiemi disgiunti si presenta nel determinare le componenti connesse di un grafo non orientato (cfr. paragrafo 5.4). La figura 22.1 (a), per esempio, mostra un grafo con quattro componenti connesse.

La seguente procedura CONNECTED-COMPONENTS usa le operazioni su insiemi disgiunti per calcolare le componenti connesse di un grafo. Una volta che la procedura preliminare CONNECTED-COMPONENTS è stata eseguita, la procedura SAME-COMPONENT può essere utilizzata per determinare se due vertici sono o no nella stessa componente connessa[1]. (L'insieme dei vertici di un grafo G è indicato con $V[G]$ e l'insieme degli archi è indicato con $E[G]$).

```
CONNECTED-COMPONENTS(G)
1  for ogni vertice v ∈ V[G]
2      do MAKE-SET(v)
3  for ogni arco (u, v) ∈ E[G]
4      do if FIND-SET(u) ≠ FIND-SET(v)
5             then UNION(u, v)

SAME-COMPONENT(u, v)
1  if FIND-SET(u) = FIND-SET(v)
2     then return TRUE
3     else retur FALSE
```

La procedura CONNECTED-COMPONENTS assegna inizialmente ad ogni vertice v un proprio insieme. Poi unisce per ogni arco (u, v) gli insiemi che contengono u e v. L'Esercizio 22.1-2 ci assicura che, dopo che tutti gli archi sono stati presi in considerazione, due vertici sono nella

[1] Nel caso in cui gli archi di un grafo siano "statici" – non cambino cioè col tempo – le componenti connesse possono essere calcolate in maniera più veloce utilizzando una visita in profondità (Esercizio 23.3-9). Alle volte però gli archi sono aggiunti "dinamicamente" e abbiamo bisogno di mantenere le componenti connesse ogni volta che aggiungiamo un arco: in questo caso, la realizzazione che diamo qui può essere più efficiente del far eseguire una nuova visita in profondità ogni volta che aggiungiamo un nuovo arco.

(a)

Arco esaminato	Collezione di insiemi disgiunti									
insiemi iniziali	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(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 grafo con quattro componenti connesse {a, b, c, d}, {e, f, g}, {h, i} e{j}.* (b) *La collezione di insiemi disgiunti dopo che ogni arco è stato esaminato.*

stessa componente connessa se e solo se i corrispondenti oggetti sono nello stesso insieme. Quindi CONNECTED-COMPONENTS crea gli insiemi in maniera tale che la procedura SAME-COMPONENT può decidere se due vertici si trovano o no nella stessa componente connessa. La figura 22.1 (b) illustra il modo in cui gli insiemi disgiunti sono costruiti da CONNECTED-COMPONENTS.

Esercizi

22.1-1 Supponiamo che CONNECTED-COMPONENTS venga eseguita sul grafo non orientato $G = (V, E)$, dove $V = \{a, b, c, d, e, f, g, h, i, j, k\}$, e gli archi di E vengano esaminati nel seguente 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). Si elenchino i vertici in ognuna delle componenti connesse dopo ogni iterazione delle linee 3-5.

22.1-2 Si mostri che dopo che tutti gli archi sono stati esaminati da CONNECTED-COMPONENTS, due vertici sono nella stessa componente connessa se e solo se sono nello stesso insieme.

22.1-3 Quante volte viene chiamata FIND-SET durante l'esecuzione di CONNECTED-COMPONENTS su un grafo non orientato $G = (V, E)$ con k componenti connesse? Quante volte è chiamata UNION? Si formulino le soluzioni in termini di $|V|$, $|E|$ e k.

22.2 Rappresentazione a lista concatenata di insiemi disgiunti

Un modo semplice per realizzare una struttura di dati per insiemi disgiunti consiste nel rappresentare ogni insieme con una lista concatenata. Il primo oggetto di ogni lista concatenata viene utilizzato come rappresentante dell'insieme. Ogni oggetto nella lista concatenata contiene un elemento dell'insieme, un puntatore all'oggetto contenuto nel successivo elemento dell'insieme e un puntatore all'indietro al rappresentante. La figura 22.2(a) mostra due insiemi. All'interno di ogni lista concatenata gli oggetti possono apparire in un qualunque ordine (anche se tale ordine deve soddisfare l'ipotesi che il primo oggetto di ogni lista sia il rappresentante).

Con questa rappresentazione mediante una lista concatenata, entrambe le operazioni MAKE-SET e FIND-SET sono semplici, e richiedono tempo $O(1)$: MAKE-SET(x) crea una nuova lista concatenata il cui unico oggetto è x; FIND-SET(x) restituisce il puntatore da x al rappresentante.

Una realizzazione semplice dell'unione

La più semplice realizzazione dell'operazione di unione che usa la rappresentazione degli insiemi come lista concatenata impiega significativamente più tempo di MAKE-SET o FIND-SET. Come mostra la figura 22.2(b), UNION(x, y) viene eseguita appendendo la lista di x alla fine della lista di y (si veda l'Esercizio 22.2-1), e il rappresentante del nuovo insieme è l'elemento che era originariamente il rappresentante dell'insieme contenente y. Sfortunatamente dobbiamo aggiornare il puntatore al rappresentante per ogni oggetto che era originariamente nella lista di x, e questo richiede tempo lineare nella lunghezza della lista di x.

In effetti non è difficile trovare una sequenza di m operazioni che richiedono tempo $\Theta(m^2)$. Siano $n = \lceil m/2 \rceil + 1$ e $q = m - n + 1 = \lfloor m/2 \rfloor$ e supponiamo di avere gli oggetti $x_1, x_2, \ldots, x_n$. Eseguiamo adesso la sequenza di $m = n + q - 1$ operazioni mostrata nella figura 22.3. Le n operazioni MAKE-SET impiegano tempo $\Theta(n)$. Dato che l'i-esima operazione di UNION aggiorna i oggetti, il numero totale di oggetti aggiornati da tutte le operazioni di UNION è

$$\sum_{i=1}^{q-1} i = \Theta(q^2) .$$

Il tempo totale impiegato è dunque $\Theta(n + q^2)$, che equivale a $\Theta(m^2)$ dal momento che $n = \Theta(m)$ e $q = \Theta(m)$; quindi in media ogni operazione richiede tempo $\Theta(m)$, perciò il tempo ammortizzato di un'operazione è $\Theta(m)$.

Un'euristica per l'unione pesata

La realizzazione della procedura UNION descritta sopra impiega tempo medio $\Theta(m)$ per chiamata, dovuto al fatto che può capitare di appendere una lista più lunga ad una più corta, e deve essere aggiornato il puntatore al rappresentante per ogni membro della lista più lunga. Supponiamo invece che ogni rappresentante includa anche la lunghezza della lista (che può essere facilmente aggiornata) e che si appenda sempre la lista più corta a quella più lunga, scegliendo in maniera arbitraria nei casi di uguaglianza. Con questa semplice ***euristica per l'unione pesata***, una singola operazione UNION può ancora impiegare tempo $\Omega(m)$ se

Figura 22.2 (a) *Rappresentazione a liste concatenate di due insiemi: uno contiene gli oggetti b, c, e, h – dove c è il rappresentante – e l'altro contiene gli oggetti d, f, g – dove f è il rappresentante. Ogni oggetto nella lista contiene un elemento dell'insieme, un puntatore all'oggetto successivo nella lista e un puntatore all'indietro al primo oggetto nella lista, che è il rappresentante.* (b) *Il risultato di* UNION(e, g)*: il rappresentante dell'insieme risultante è f.*

Operazioni	Numero di oggetti aggiornati
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
⋮	⋮
UNION(x_{q-1}, x_q)	$q - 1$

Figura 22.3 *Una sequenza di m operazioni che impiega tempo $\Theta(m^2)$ usando la rappresentazione a liste concatenate e la realizzazione più semplice di* UNION*: in quest'esempio $n = \lceil m/2 \rceil + 1$ e $q = m - n + 1$.*

entrambi gli insiemi hanno $\Omega(m)$ elementi. Come mostra però il seguente teorema, una sequenza di m operazioni MAKE-SET, UNION e FIND-SET, di cui n sono MAKE-SET, impiega tempo $O(m + n \lg n)$.

Teorema 22.1

Usando la rappresentazione degli insiemi disgiunti a liste concatenate e l'euristica per l'unione pesata, una sequenza di m operazioni MAKE-SET, UNION e FIND-SET, di cui n sono MAKE-SET, impiega tempo $O(m + n \lg n)$.

Dimostrazione. Cominciamo calcolando, per ogni oggetto in un insieme di dimensione n, un limite superiore al numero di volte in cui il puntatore al rappresentante di un oggetto è stato aggiornato. Consideriamo un dato oggetto x. Sappiamo che ogni volta che il puntatore al rappresentante di x è stato aggiornato, x doveva trovarsi nell'insieme più piccolo. Dunque la prima volta che il puntatore al rappresentante di x è stato aggiornato, l'insieme risultante deve aver avuto almeno due elementi. Analogamente, la volta successiva in cui il puntatore al rappresentante di x è stato aggiornato, l'insieme risultante doveva aver avuto al minimo quattro elementi. Procedendo in questo modo, ci rendiamo conto che per ogni $k \leq n$, dopo che il puntatore al rappresentante di x è stato aggiornato $\lceil \lg k \rceil$ volte, l'insieme risultante deve aver avuto almeno k elementi. Dato che l'insieme più grande ha al più n elementi, il puntatore al rappresentante di ogni oggetto può essere stato aggiornato al più $\lceil \lg n \rceil$ volte fra tutte le operazioni di UNION. Il tempo totale usato nell'aggiornare gli n oggetti è dunque $O(n \lg n)$.

Da ciò ricaviamo facilmente il tempo necessario per l'intera sequenza di m operazioni: ogni operazione MAKE-SET e FIND-SET impiega tempo $O(1)$, e se ne fanno al più $O(m)$. Il tempo totale per l'intera sequenza è dunque $O(m + n \lg n)$. ■

Esercizi

22.2-1 Si scrivano delle procedure per MAKE-SET, FIND-SET e UNION che utilizzino la rappresentazione a liste concatenate e l'euristica per l'unione pesata. Si assuma che ogni oggetto x abbia gli attributi $rep[x]$ (che punta al rappresentante dell'insieme contenente x), $last[x]$ (che punta all'ultimo oggetto nella lista concatenata che contiene x) e $size[x]$ (che riporta la dimensione dell'insieme che contiene x). Si può assumere che $last[x]$ e $size[x]$ siano corretti solo se x è un rappresentante. In che modo è possibile concatenare la lista di x a quella di y per l'operazione UNION(x, y) senza usare $last[y]$ e senza scorrere tutta la lista di y?

22.2-2 Si mostrino la struttura di dati risultante e le risposte restituite dalle operazioni FIND-SET nel programma seguente, usando la rappresentazione a liste concatenate con l'euristica per l'unione pesata.

```
for i ← 1 to 16
    do MAKE-SET(x_i)
for i ← 1 to 15 by 2
    do UNION(x_i, x_{i+1})
for i ← 1 to 13 by 4
    do UNION(x_i, x_{i+2})
UNION(x_1, x_5)
UNION(x_{11}, x_{13})
UNION(x_1, x_{10})
FIND-SET(x_2)
FIND-SET(x_9)
```

22.2-3 Si deduca, modificando la dimostrazione del Teorema 22.1, che si possono ottenere dei limiti di tempo ammortizzato di $O(1)$ per MAKE-SET e FIND-SET e di $O(\lg n)$ per UNION usando la rappresentazione a liste concatenate e l'euristica per l'unione pesata.

Figura 22.4 *Una foresta di insiemi disgiunti.* (**a**) *Due alberi che rappresentano i due insiemi di figura 22.2. L'albero sulla sinistra rappresenta l'insieme* $\{b, c, e, h\}$, *dove* c *è il rappresentante, e l'albero sulla destra rappresenta l'insieme* $\{d, f, g\}$, *dove* f *è il rappresentante.* (**b**) *Il risultato di* UNION(e, g).

22.2-4 Si fornisca un limite asintotico stretto sul tempo di esecuzione della sequenza di operazioni di figura 22.3, assumendo la rappresentazione a liste concatenate e l'euristica per l'unione pesata.

22.3 Foreste di insiemi disgiunti

In una realizzazione più veloce degli insiemi disgiunti, gli insiemi sono rappresentati da alberi radicati, in cui ogni nodo contiene un elemento e ogni albero rappresenta un insieme. In una ***foresta di insiemi disgiunti***, illustrata nella figura 22.4(a), ogni elemento ha un puntatore solo al padre. La radice di ogni albero contiene il rappresentante ed è padre di se stesso. Come vedremo, nonostante che gli algoritmi intuitivi che usano questa rappresentazione non siano più veloci di quelli che usano la rappresentazione a liste concatenate, con l'introduzione di due euristiche – l'"unione per rango" e la "compressione dei cammini" – possiamo ottenere le strutture di dati per insiemi disgiunti asintoticamente più veloci conosciute.

Realizziamo le tre operazioni per insiemi disgiunti nel modo seguente: una operazione MAKE-SET crea semplicemente un albero con un unico nodo; un'operazione FIND-SET viene eseguita controllando i puntatori al padre fino a quando non viene trovata la radice dell'albero (i nodi visitati lungo questo cammino verso la radice costituiscono il ***cammino di accesso***); un'operazione di UNION, mostrata nella figura 22.4(b), fa sì che la radice di un albero punti alla radice dell'altro.

Euristiche per migliorare il tempo di esecuzione

Finora non abbiamo apportato miglioramenti alla realizzazione con liste concatenate: una sequenza di $n - 1$ operazioni UNION può creare un albero che non è altro che una catena lineare di n nodi. Usando due euristiche, invece, riusciamo a ottenere un tempo di esecuzione che è quasi lineare sul numero totale m di operazioni.

La prima euristica, ***unione per rango***, è simile all'euristica per l'unione pesata che abbiamo usato nella rappresentazione a liste concatenate. L'idea è di far sì che la radice dell'albero più basso punti alla radice dell'albero più alto. Invece di tener traccia esplicitamente della dimensione del sottoalbero di cui un nodo è radice, useremo un approccio che facilita l'analisi. Per ogni nodo manterremo un ***rango*** che è un limite superiore all'altezza del nodo;

Figura 22.5 *Compressione dei cammini durante l'operazione* FIND-SET: *le frecce e i cicli alle radici sono omessi.* (a) *Un albero che rappresenta un insieme prima dell'esecuzione di* FIND-SET(a): *i triangoli rappresentano sottoalberi dei quali vengono mostrati i nodi corrispondenti alle radici; ogni nodo ha un puntatore al padre.* (b) *Lo stesso insieme dopo l'esecuzione di* FIND-SET(a): *ogni nodo sul cammino d'accesso adesso punta direttamente alla radice.*

con l'unione per rango, la radice col rango più piccolo viene fatta puntare alla radice con rango maggiore nel corso di un'operazione UNION.

Anche la seconda euristica, ***compressione dei cammini***, è abbastanza semplice e molto efficace. Come mostra la figura 22.5, viene usata nel corso delle operazioni FIND-SET per far sì che ogni nodo sul cammino di accesso punti direttamente alla radice. La compressione dei cammini non cambia alcun rango.

Procedure per foreste di insiemi disgiunti

Per realizzare una foresta di insiemi disgiunti che utilizzi l'euristica dell'unione per rango, dobbiamo poter memorizzare i ranghi. Associamo a ogni nodo x il valore intero $rank[x]$, che è un limite superiore all'altezza di x (il numero di archi nel cammino più lungo fra x e una foglia sua discendente). Quando viene creato un insieme di un unico elemento con MAKE-SET, il rango iniziale dell'unico nodo nell'albero corrispondente è 0. L'esecuzione di FIND-SET non cambia i ranghi, mentre ogni volta che viene applicata UNION a due alberi, rendiamo la radice con il rango maggiore padre di una radice con rango minore: in caso di uguaglianza, scegliamo in maniera casuale una delle radici come padre e incrementiamo il suo rango.

Cerchiamo di scrivere una procedura che realizzi questo metodo. Indichiamo con $p[x]$ il padre del nodo x. La procedura LINK, una sottoprocedura invocata da UNION, utilizza come input due puntatori alle radici dei due alberi.

```
MAKE-SET(x)
  p[x] ← x
  rank[x] ← 0
```



```
UNION(x, y)
  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
  if rank[x] > rank[y]
    then p[y] ← x
    else p[x] ← y
         if rank[x] = rank[y]
           then rank[y] ← rank[y] + 1
```

La procedura FIND-SET con compressione dei cammini è abbastanza semplice:

```
FIND-SET(x)
1  if x ≠ p[x]
2     then p[x] ← FIND-SET(p[x])
3  return p[x]
```

La procedura FIND-SET è un ***metodo a due passate***: nella prima risale il cammino di accesso per cercare la radice, e poi lo discende per aggiornare tutti i nodi in maniera tale che puntino direttamente alla radice. Ogni chiamata di FIND-SET (x) restituisce $p[x]$ nella linea 3. Se x è la radice, allora la linea 2 non è eseguita e viene restituito $p[x] = x$: è questo il caso in cui la ricorsione termina. Altrimenti la linea 2 viene eseguita, e la chiamata ricorsiva con parametro $p[x]$ restituisce un puntatore alla radice. La linea 2 aggiorna il nodo x facendolo puntare direttamente alla radice, restituendo poi il puntatore nella linea 3.

Effetto delle euristiche sul tempo di esecuzione

Separatamente, sia l'unione per rango che la compressione dei cammini migliorano il tempo di esecuzione delle operazioni su foreste di insiemi disgiunti, e tale miglioramento è anche maggiore quando le due euristiche sono usate assieme. Da sola, l'unione per rango produce lo stesso tempo di esecuzione che ottenevamo con l'euristica per l'unione pesata nel caso della rappresentazione a liste: la realizzazione risultante viene eseguita in tempo $O(m \lg n)$ (cfr. Esercizio 22.4-3), e questo limite è stretto (cfr. Esercizio 22.3-3). Benché non possiamo provarlo qui, se abbiamo n operazioni MAKE-SET (e dunque al più $n - 1$ operazioni UNION) e f operazioni MAKE-SET, l'euristica per la compressione dei cammini ottiene da sola un tempo di esecuzione nel caso peggiore di $\Theta(f \log_{1+f/n} n)$ se $f \geq n$ e di $\Theta(n + f \lg n)$ se $f < n$.

Quando vengono usate sia l'unione per rango che la compressione dei cammini, il tempo di esecuzione nel caso pessimo diventa $O(m\alpha(m, n))$, dove $\alpha(m, n)$ è l'inversa della funzione di Ackermann (che cresce *molto* lentamente) che definiremo nel paragrafo 22.4. Per qualunque applicazione concepibile di una struttura di dati per insiemi disgiunti, $\alpha(m, n) \leq 4$, e quindi il tempo di esecuzione può essere visto come lineare in m in tutte le situazioni pratiche. Nel paragrafo 22.4 proveremo il limite leggermente più debole di $O(m \lg^* n)$.

Esercizi

22.3-1 Si risolva nuovamente l'Esercizio 22.2-2 usando una foresta di insiemi disgiunti con l'unione per rango e la compressione dei cammini.

22.3-2 Si scriva una versione non ricorsiva di FIND-SET con la compressione dei cammini.

22.3-3 Si fornisca una sequenza di m operazioni MAKE-SET, UNION e FIND-SET – di cui n siano operazioni MAKE-SET – che impiega tempo $\Omega(m \lg n)$ quando viene usata soltanto l'unione per rango.

* *22.3-4* Si dimostri che ogni sequenza di m operazioni MAKE-SET, FIND-SET e LINK, dove tutte le operazioni LINK appaiono prima di tutte le operazioni FIND-SET, impiega soltanto tempo $O(m)$ se vengono usate sia la compressione dei cammini che l'unione per rango. Cosa accade nella stessa situazione se viene usata soltanto l'euristica per la compressione dei cammini?

* 22.4 Analisi dell'unione per rango con compressione dei cammini

Come visto nel paragrafo 22.3, il tempo di esecuzione nel caso di uso combinato delle euristiche per l'unione per rango e per la compressione dei cammini è $O(m\alpha(m, n))$ per m operazioni su insiemi disgiunti con n elementi. In questo paragrafo esamineremo la funzione α mostrando quanto sia lenta la sua crescita. Dopodiché, invece che presentare la dimostrazione molto complessa del tempo di esecuzione $O(m\alpha(m, n))$, ne daremo una più semplice per un limite superiore leggermente più debole sul tempo di esecuzione, dato da $O(m \lg^* n)$.

La funzione di Ackermann e la sua inversa

Per meglio capire la funzione di Ackermann e la sua inversa α, può essere d'aiuto utilizzare una notazione per l'esponenziazione ripetuta. Dato un intero $i \geq 0$, l'espressione

$$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}i$$

indica la funzione $g(i)$ definita ricorsivamente da

$$g(i) = \begin{cases} 2 & \text{se } i = 0 , \\ 2^{g(i-1)} & \text{se } i \geq 1 . \end{cases}$$

Intuitivamente, il parametro i fornisce la "altezza della pila di 2" che compone l'esponente.

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 1$	2^1	2^2	2^3	2^4
$i = 2$	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i = 3$	2^{2^2}	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}16$	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}16$	$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}16$

Figura 22.6 *I valori di $A(i, j)$ per valori piccoli di i e di j.*

Per esempio

$$2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}4 = 2^{2^{2^{2^2}}} = 2^{65536} .$$

Rammentiamo adesso la definizione della funzione $\lg^*$ (Capitolo 2) in termini della funzione $\lg^{(i)}$ definita per l'intero $i \geq 0$:

$$\lg^{(i)} n = \begin{cases} n & \text{se } i = 0 , \\ \lg(\lg^{(i-1)} n) & \text{se } i > 0 \text{ e } \lg^{(i-1)} n > 0 , \\ \text{indefinito} & \text{se } i > 0 \text{ e } \lg^{(i-1)} n \leq 0 \text{ o } \lg^{(i-1)} n \text{ è indefinito} \end{cases}$$

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

La funzione $\lg^*$ è fondamentalmente l'inversa dell'esponenziazione ripetuta:

$$\lg^* 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}n = n + 1$$

Possiamo ora finalmente introdurre la *funzione di Ackermann*, che è definita sugli interi positivi i e j come

$$\begin{aligned} A(1, j) &= 2^j && \text{per } j \geq 1 , \\ A(i, 1) &= A(i-1, 2) && \text{per } i \geq 2 , \\ A(i, j) &= A(i-1, A(i, j-1)) && \text{per } i, j \geq 2 . \end{aligned}$$

La figura 22.6 mostra il valore della funzione $A(i, j)$ per valori piccoli di i e j.

La figura 22.7 mostra schematicamente perché la funzione di Ackermann abbia una tale crescita esplosiva. La prima riga, esponenziale nel numero della colonna j, cresce già rapidamente. La seconda riga consiste del sottoinsieme delle colonne della prima riga $2^2, 2^{2^2}, 2^{2^{2^2}}, \ldots$, con una grossa distanza fra gli elementi. Le linee fra righe adiacenti indicano le colonne nella riga con il numero minore che sono nel sottoinsieme incluso nella riga con il numero più grande.

La terza riga consiste del sottoinsieme $2^{2^2}, 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}16, 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}16, \ldots$ delle colonne della seconda riga, i cui elementi hanno una distanza sempre maggiore e che è un sottoinsieme

Figura 22.7 *La crescita esplosiva della funzione di Ackermann: le linee fra le righe $i-1$ e i indicano gli elementi della riga $i-1$ che compaiono nella riga i. A causa della crescita esplosiva la spaziatura orizzontale non può essere riprodotta in scala. La spaziatura orizzontale tra i valori della riga $i-1$ che compaiono nella riga i aumenta moltissimo con i numeri di riga e di colonna: se uniamo gli elementi nella riga i con i corrispondenti nella riga 1, la crescita esplosiva risulta ancora più evidente.*

ancora più sparso delle colonne della prima riga. In generale, la distanza fra le colonne della riga $i-1$ che appaiono nella riga i aumenta incredibilmente sia con il numero della colonna che con il numero della riga. Si osservi che $A(2, j) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}j}$ per tutti gli interi $j \geq 1$. Quindi, per $i > 2$, la funzione $A(i, j)$ cresce ancora più rapidamente di $2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}j}$.

Definiamo l'inversa della funzione di Ackermann nel seguente modo[2]:

$$\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) > \lg n\} .$$

Fissato un valore di n, la funzione $\alpha(m, n)$ è monotonicamente decrescente al crescere di m. Per rendersi conto di questa proprietà, basta notare che $\lfloor m/n \rfloor$ cresce monotonicamente al crescere di m; quindi, dal momento che n è fissato, il più piccolo valore di i che è necessario per far superare a $A(i, \lfloor m/n \rfloor)$ il valore di $\lg n$ decresce in maniera monotona. Questa proprietà corrisponde all'intuizione che sta sotto alle foreste di insiemi disgiunti con compressione dei cammini: dato un certo numero n di elementi diversi, ci aspetteremmo che la lunghezza media del cammino di accesso decresca al crescere del numero di operazioni m a causa della compressione dei cammini. Se eseguiamo m operazioni in tempo $O(m\alpha(m, n))$, allora il tempo medio per operazione è $O(\alpha(m, n))$, che decresce monotonicamente al crescere di m.

A supporto della nostra affermazione precedente che $\alpha(m, n) \leq 4$ per tutti gli scopi pratici, cominciamo con il notare che il valore di $\lfloor m/n \rfloor$ è almeno 1, dato che $m \geq n$. Visto che la funzione di Ackermann è strettamente crescente in ogni argomento, $\lfloor m/n \rfloor \geq 1$ implica $A(i, \lfloor m/n \rfloor) \geq A(i, 1)$ per ogni $i \geq 1$. In particolare, $A(4, \lfloor m/n \rfloor) \geq A(4, 1)$; ma dato che abbiamo

$$\begin{aligned} A(4, 1) &= A(3, 2) \\ &= 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}16} . \end{aligned}$$

[2] Benché in senso strettamente matematico questa funzione non sia l'inverso della funzione di Ackermann, cattura lo spirito dell'inverso nella sua crescita, che è tanto lenta quanto quella della funzione di Ackermann è veloce. La ragione per cui si utilizza la misteriosa soglia $\lg n$ è chiarita nella dimostrazione del tempo di esecuzione $O(m\alpha(m, n))$, che non è riportata in questo libro.

che è di gran lunga più grande del numero stimato di atomi nell'universo osservabile (circa 10^{80}), è solo per valori di n eccessivamente grandi che $A(4, 1) \leq \lg n$, e quindi $\alpha(m, n) \leq 4$ per tutti gli scopi pratici. Si noti che il limite $O(m \lg^* n)$ è solo leggermente più debole del limite $O(m\alpha(m, n))$: $\lg^* 2^{65536} = 5$, per cui $\lg^* n \leq 5$ per tutti gli scopi pratici.

Proprietà dei ranghi

Nel seguito del paragrafo proveremo il limite $O(m \lg^* n)$ sul tempo di esecuzione delle operazioni su insiemi disgiunti con unione per rango e compressione dei cammini. Per far ciò, cominceremo col dimostrare alcune semplici proprietà dei ranghi.

Lemma 22.2

Per tutti i nodi x abbiamo che $rank[x] \leq rank[p[x]]$, dove la disuguaglianza è stretta nel caso $x \neq p[x]$. Il valore di $rank[x]$ è inizialmente 0 e cresce col tempo fintanto che x è una radice, ossia $x = p[x]$, dopodiché, $rank[x]$ non cambia. Il valore di $rank[p[x]]$ è una funzione che cresce monotonicamente con il tempo (fissato x, $p[x]$ può cambiare nel tempo).

Dimostrazione. La dimostrazione procede con un'induzione immediata sul numero di operazioni, usando le realizzazioni di MAKE-SET, UNION e FIND-SET che appaiono nel paragrafo 22.3. La lasciamo al lettore nell'Esercizio 22.4-1. ■

Indichiamo con size(x) il numero dei nodi nell'albero che ha x come radice, contando il nodo x stesso.

Lemma 22.3

Per tutte le radici x di alberi, size(x) $\geq 2^{rank[x]}$.

Dimostrazione. La dimostrazione procede per induzione sul numero di operazioni LINK. Si noti che le operazioni FIND-SET non cambiano né il rango di una radice né la dimensione del suo sottoalbero.

Base: il lemma è vero prima dell'occorrenza della prima operazione LINK, dato che i ranghi sono inizialmente 0 e ogni albero contiene un nodo.

Passo induttivo: assumiamo che il lemma valga prima di eseguire l'operazione LINK(x, y). Sia *rank* il rango prima dell'esecuzione di LINK, e sia *rank'* subito dopo: definiamo size e size' in maniera analoga.

Se $rank[x] \neq rank[y]$, possiamo assumere senza perdita di generalità che $rank[x] < rank[y]$. Il nodo y è la radice dell'albero che si forma dopo l'operazione LINK, e

$$\begin{aligned} \text{size}'(y) &= \text{size}(x) + \text{size}(y) \\ &\geq 2^{rank[x]} + 2^{rank[y]} \\ &\geq 2^{rank[y]} \\ &= 2^{rank'[y]} . \end{aligned}$$

Tranne che per il nodo y, non cambiano ranghi e dimensioni.

Se $rank[x] = rank[y]$, il nodo y è ancora la radice del nuovo albero, e

$$\begin{aligned} size'(y) &= size(x) + size(y) \\ &\geq 2^{rank[x]} + 2^{rank[y]} \\ &= 2^{rank[y]+1} \\ &= 2^{rank'[y]} . \end{aligned}$$

■

Lemma 22.4

Per ogni intero $r \geq 0$, esistono al più $n/2^r$ nodi di rango r.

Dimostrazione. Sia dato un certo valore di r. Supponiamo che quando assegniamo un rango r ad un nodo x (nella linea 2 di MAKE-SET o nella linea 5 di LINK), attacchiamo un'etichetta x a ogni nodo nell'albero che ha come radice x. Per il lemma 22.3 ogni volta vengono etichettati almeno 2^r nodi. Supponiamo che la radice dell'albero che contiene il nodo x cambi: il lemma 22.2 ci assicura che il rango della nuova radice (o, in effetti, di ogni antenato proprio di x) è almeno $r + 1$. Dato che etichettiamo solo quando a una radice viene assegnato il rango r, nessun nodo in questo nuovo albero sarà mai etichettato. Quindi ogni nodo è etichettato al più una volta, quando alla sua radice è assegnato per la prima volta il rango r. Dato che ci sono n nodi, possono esistere al più n nodi etichettati, con almeno 2^r etichette assegnate per ogni nodo di rango r. Se ci fossero più di $n/2^r$ nodi di rango r, allora più di $2^r \cdot (n/2^r) = n$ nodi sarebbero etichettati da un nodo di rango r, il che porta a una contraddizione. Dunque, dato un rango r, questo viene assegnato al più a $n/2^r$ nodi. ■

Corollario 22.5

Ogni nodo ha un rango al più $\lfloor \lg n \rfloor$.

Dimostrazione. Se fosse $r > \lg n$, esisterebbero al più $n/2^r < 1$ nodi di rango r. Dato che i ranghi sono numeri naturali, il corollario segue immediatamente. ■

Verifica del limite di tempo

Useremo il metodo degli aggregati per l'analisi ammortizzata (cfr. paragrafo 18.1) per dimostrare che il limite di tempo è $O(m \lg^* n)$. Nell'eseguire l'analisi ammortizzata, conviene assumere che si invochi l'operazione LINK invece dell'operazione UNION. Detto questo, dal momento che i parametri della procedura LINK sono puntatori alle due radici, assumeremo che, ogni volta che sia necessario, vengano eseguite le operazioni FIND-SET. Il seguente lemma mostra che il tempo di esecuzione asintotico non cambia anche se si considerano le ulteriori operazioni FIND-SET.

Lemma 22.6

Supponiamo di convertire una sequenza S' di m' operazioni MAKE-SET, UNION e FIND-SET in una sequenza S di m operazioni MAKE-SET, LINK e FIND-SET facendo diventare ogni UNION due operazioni FIND-SET seguite da un LINK. Se la sequenza S viene eseguita in tempo $O(m \lg^* n)$, allora la sequenza S' viene eseguita in tempo $O(m' \lg^* n)$.

Dimostrazione. Dato che ogni operazione UNION nella sequenza S' è convertita in tre operazioni in S, abbiamo che $m' \leq m \leq 3m'$. Dato che $m = O(m')$, un limite di tempo $O(m \lg^* n)$ per la sequenza S implica un limite di tempo $O(m' \lg^* n)$ per la sequenza originale S'. ■

Nel seguito del paragrafo assumeremo che la sequenza di partenza di m' operazioni MAKE-SET, UNION e FIND-SET sia stata convertita in una sequenza di m operazioni MAKE-SET, LINK e FIND-SET. Dimostreremo adesso un limite di tempo $O(m \lg^* n)$ per la sequenza derivata, e utilizzeremo poi il lemma 22.6 per dimostrare il tempo di esecuzione $O(m' \lg^* n)$ della sequenza originale di m' operazioni.

Teorema 22.7

Una sequenza di m operazioni MAKE-SET, LINK e FIND-SET, di cui n sono operazioni MAKE-SET, può essere eseguita su una foresta di insiemi disgiunti con unione per rango e compressione dei cammini in tempo $O(m \lg^* n)$ nel caso pessimo.

Dimostrazione. Assegneremo ***unità di prezzo*** che corrispondono al costo reale di ogni operazione sugli insiemi e calcoleremo la somma totale delle unità di prezzo assegnate una volta che l'intera sequenza di operazioni sia stata eseguita. Questo totale rappresenterà il costo reale di tutte le operazioni.

I prezzi assegnati alle operazioni MAKE-SET e LINK sono facili: una unità per operazione. Dato che queste operazioni impiegano ognuna tempo reale $O(1)$, i prezzi assegnati coincidono con i costi reali delle operazioni.

Prima di discutere i prezzi assegnati alle operazioni FIND-SET, partizioneremo i ranghi dei nodi in ***blocchi*** assegnando il rango r al blocco $\lg^* r$ per $r = 0, 1, \ldots, \lfloor \lg n \rfloor$. Dato che $\lfloor \lg n \rfloor$ è il rango massimo, il blocco con il numero più alto è dunque il blocco $\lg^*(\lg n) = \lg^* n - 1$. Per alleggerire la notazione, definiremo, dato un intero $j \geq -1$,

$$B(j) = \begin{cases} -1 & \text{se } j = -1 , \\ 1 & \text{se } j = 0 , \\ 2 & \text{se } j = 1 , \\ 2^{2^{\cdot^{\cdot^{2}}}} \Big\}_{j-1} & \text{se } j \geq 2 . \end{cases}$$

Dunque per $j = 0, 1, \ldots, \lg^* n - 1$, il blocco j-esimo consiste dell'insieme dei ranghi

$$\{B(j-1) + 1, B(j-1) + 2, \ldots, B(j)\}.$$

Useremo due tipi di prezzi per un'operazione FIND-SET: ***unità di blocco*** e ***unità di cammino***. Supponiamo che FIND-SET parta dal nodo x_0, e che il cammino di accesso consista dei nodi $x_0, x_1, \ldots, x_l$, dove per $j = 1, 2, \ldots, l$ il nodo x_j è $p[x_{j-1}]$ e x_l è una radice e dunque coincide con $p[x_l]$. Per $j = 0, 1, \ldots, \lg^* n - 1$ assegniamo una unità di blocco all'*ultimo* nodo del cammino il cui rango appartiene al blocco j. (Si faccia attenzione che il lemma 22.2 implica che, su ogni cammino di accesso, i nodi con ranghi nello stesso blocco sono consecutivi). Assegneremo inoltre una unità di blocco al figlio della radice, ovverossia a x_{l-1}. Dato che i ranghi sono strettamente crescenti lungo ogni cammino di accesso, una formulazione equivalente assegna una unità di blocco a ogni nodo x_j tale che $p[x_j] = x_l$ (x_j è la radice o un suo figlio) o $\lg^* rank[x_j] < \lg^* rank[x_{j+1}]$ (il blocco cui appartiene il rango di x_j è diverso da quello del padre). Assegniamo poi una unità di cammino ad ogni nodo sul cammino di accesso a cui non assegniamo una unità di blocco.

Una volta che a un nodo diverso dalla radice o da un suo figlio viene assegnata una unità di blocco, non gli verrà mai più assegnata una unità di cammino. Per capire la ragione, si osservi che ogni volta che avviene la compressione di un cammino, il rango di un nodo x_i per il quale $p[x_i] \neq x_i$ rimane lo stesso, mentre il nuovo padre di x_i ha un rango strettamente maggiore di quello del vecchio. La differenza fra il rango di x_i e quello del padre è una funzione monotonicamente crescente con il tempo. Quindi la differenza tra $\lg^* rank[p[x_i]]$ e $\lg^* rank[x_i]$ è a sua volta una funzione monotonicamente crescente con il tempo. Non appena accade che x_i e il padre hanno il rango in blocchi diversi, avranno sempre rango in blocchi diversi, e quindi a x_i non verrà mai più assegnata una unità di cammino.

Dato che abbiamo assegnato prezzi una sola volta per ogni nodo visitato in ognuna delle operazioni FIND-SET, il numero totale delle unità assegnate coincide con il numero totale dei nodi visitati durante le operazioni FIND-SET: questo totale rappresenta il costo reale di tutte le operazioni FIND-SET. Mostreremo che questo totale è $O(m \lg^* n)$.

Il numero delle unità di blocco è facile da limitare: viene assegnata al più una unità di blocco per ogni numero di blocco su un dato cammino di accesso, più una per il figlio della radice, e dato che i numeri di blocco variano da 0 a $\lg^* n - 1$, vengono assegnate al più $\lg^* n + 1$ unità di blocco per ogni operazione FIND-SET. Dunque per tutte le operazioni FIND-SET vengono assegnate al più $m(\lg^* n + 1)$ unità di blocco.

Limitare le unità di cammino è un po' più difficile. Inizieremo osservando che se a un nodo x_i è assegnata una unità di cammino, allora $p[x_i] \neq x_i$ prima della compressione del cammino, e quindi a x_i verrà assegnato un nuovo padre nel corso della compressione del cammino. Inoltre, come avevamo osservato, il nuovo padre di x_i ha un rango più alto di quello del vecchio. Supponiamo che il rango di x_i sia nel blocco j. Quante volte può essere assegnato a x_i un nuovo padre, e quindi assegnata una unità di cammino, prima che a x_i sia assegnato un padre il cui rango è in un blocco differente (dopodiché a x_i non verrà mai più assegnata una unità di cammino)? Questo numero di volte può essere massimizzato se x_i ha il rango più basso nel suo blocco, sia esso $B(j-1)+1$, e il rango dei suoi genitori assume successivamente i valori $B(j-1)+2, B(j-1)+3, \ldots, B(j)$. Dato che esistono $B(j) - B(j-1) - 1$ ranghi del genere, ne concludiamo che a un vertice possono essere assegnate al più $B(j) - B(j-1) - 1$ unità di cammino fintantoché il suo rango è nel blocco j.

Il passo successivo per limitare le unità di cammino consiste nel limitare il numero di nodi che hanno il rango nel blocco j per un dato intero $j \geq 0$ (si ricordi che, per il lemma 22.2, il rango di un nodo è fissato non appena diventa figlio di un altro nodo). Indichiamo con $N(j)$ il numero di nodi il cui rango appartiene al blocco j: per il lemma 22.4 avremo

$$N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r} .$$

Poiché n è il numero totale dei nodi, dovrà essere anche $N(j) \leq n$. Nel caso $j = 0$, ricordando che $B(0) = 1$, avremo $N(0) \leq n = n/B(0)$, mentre per $j \geq 1$

$$\begin{aligned} N(j) &\leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-B(j-1)-1} \frac{1}{2^r} \\ &< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r} \\ &= \frac{n}{2^{B(j-1)}} \\ &= \frac{n}{B(j)} . \end{aligned}$$

Dunque $N(j) \leq n/B(j)$ per tutti gli interi $j \geq 0$.

Terminiamo limitando le unità di cammino ottenute sommando per tutti i blocchi il prodotto del numero massimo di nodi con rango in un dato blocco e il numero massimo di unità di cammino per nodo in quel blocco. Indicando con $P(n)$ il numero totale di unità di cammino, avremo

$$\begin{aligned} P(n) &\leq \sum_{j=0}^{\lg^* n - 1} \frac{n}{B(j)} (B(j) - B(j-1) - 1) \\ &\leq \sum_{j=0}^{\lg^* n - 1} \frac{n}{B(j)} \cdot B(j) \\ &= n \lg^* n . \end{aligned}$$

Perciò il numero totale di unità aggiunte dalle operazioni FIND-SET è $O(m(\lg^* n + 1) + n \lg^* n)$, che è $O(m \lg^* n)$ dato che $m \geq n$. Poiché abbiamo $O(n)$ operazioni MAKE-SET e LINK, ognuna delle quali assegna una unità, il tempo totale è $O(m \lg^* n)$. ■

Corollario 22.8

Una sequenza di m operazioni MAKE-SET, UNION e FIND-SET, di cui n sono operazioni MAKE-SET, può essere eseguita su una foresta di insiemi disgiunti con unione per rango e compressione di cammini in tempo $O(m \lg^* n)$ nel caso pessimo.

Dimostrazione. Immediata per il Teorema 22.7 e il lemma 22.6. ■

Esercizi

22.4-1 Dimostrare il lemma 22.2.

22.4-2 Quanti bit sono necessari a un nodo x per memorizzare size(x)? Quanti per $rank[x]$?

22.4-3 Si fornisca una dimostrazione semplice, usando il lemma 22.2 e il corollario 22.5, che le operazioni su una foresta di insiemi disgiunti con unione per rango ma senza compressione dei cammini impiegano tempo $O(m \lg n)$.

* **22.4-4** Supponiamo di modificare la regola sull'assegnamento delle unità di prezzo in maniera tale che si assegni una unità di blocco all'ultimo nodo del cammino di accesso il cui rango appartiene al blocco j per $j = 0, 1, \ldots, \lg^* n - 1$, mentre se ciò non avviene gli si assegni una unità di cammino: dunque se un nodo è un figlio della radice e non è l'ultimo nodo del blocco, gli viene assegnata una unità di cammino, non una unità di blocco. Dimostrate che ad un dato nodo potrebbero essere assegnate $\Omega(m)$ unità di cammino mentre il suo rango appartiene a un dato blocco j.

Problemi

22-1 *Minimo off-line*

Il ***problema del minimo off-line*** concerne il mantenimento di un insieme dinamico P di elementi dal dominio $\{1, 2, \ldots, n\}$ dopo l'esecuzione delle operazioni INSERT ed EXTRACT-MIN. Iniziamo con una sequenza S di n chiamate a INSERT e m chiamate a EXTRACT-MIN, dove ogni chiave in $\{1, 2, \ldots, n\}$ è inserita esattamente una volta: vorremmo determinare quale chiave è restituita da ogni chiamata di EXTRACT-MIN. Più in particolare, vorremmo inserire tali chiavi in un array *extracted*$[1 .. m]$ dove, per $i = 1, 2, \ldots, m$, *extracted*$[i]$ è la chiave restituita dall'i-esima chiamata di EXTRACT-MIN. Il problema è "off-line" nel senso che possiamo eseguire l'intera sequenza S prima di determinare anche una sola delle chiavi restituite.

a. Nell'istanza seguente del problema del minimo off-line, ogni INSERT è rappresentata da un numero mentre ogni EXTRACT-MIN è rappresentata dalla lettera E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.

Inserite i valori corretti nell'array *extracted*.

Per realizzare un algoritmo per questo problema, spezziamo la sequenza S in sottosequenze omogenee: rappresentiamo cioè S come

$I_1, E, I_2, E, I_3, \ldots, I_m, E, I_{m+1}$,

dove ogni E rappresenta una singola chiamata di EXTRACT-MIN e ogni I_j rappresenta una sequenza (eventualmente vuota) di chiamate di INSERT. Per ogni sottosequenza I_j poniamo inizialmente le chiavi inserite da queste operazioni in un insieme K_j, che è vuoto se I_j è vuoto. Poi eseguiamo la seguente procedura

```
OFF-LINE-MINIMUM(m, n)
for i ← 1 to n
    do determina j tale che i ∈ K_j
        if j ≠ m + 1
            then extracted[j] ← i
                sia l il valore minimo più grande di j tale che l'insieme K_l esiste
                K_l ← K_l ∪ K_j, dopodiché distrugge K_j
return extracted
```

b. Si dimostri che l'array *extracted* restituito da OFF-LINE-MINIMUM è corretto.

c. Si descriva come usare una struttura di dati per insiemi disgiunti al fine di realizzare OFF-LINE-MINIMUM in maniera efficiente, e si dia un limite stretto sul tempo di esecuzione nel caso peggiore della realizzazione proposta.

22-2 *Determinazione della profondità*

Nel ***problema della determinazione della profondità*** vogliamo mantenere una foresta $\mathcal{F} = \{T_i\}$ di alberi radicati durante l'esecuzione di tre operazioni:

MAKE-TREE(v) crea un albero il cui unico nodo è v.

FIND-DEPTH(v) restituisce la profondità di un nodo v all'interno del suo albero.

GRAFT(r, v) fa diventare il nodo r, che si assume essere la radice di un albero, il figlio del nodo v, che si suppone essere in un albero differente da quello di r, anche se può non essere una radice.

a. Immaginiamo di usare una rappresentazione per gli alberi simile a una foresta di insiemi disgiunti: $p[v]$ è il padre del nodo v, e $p[v] = v$ se v è una radice. Si dimostri che, se realizziamo GRAFT (r, v) con l'assegnamento $p[r] \leftarrow v$ e realizziamo FIND-DEPTH(v) seguendo il cammino di accesso fino alla radice e restituendo il totale di tutti i nodi incontrati diversi da v, il tempo di esecuzione nel caso pessimo di una sequenza di m operazioni MAKE-TREE, FIND-DEPTH e GRAFT è $\Theta(m^2)$.

Possiamo ridurre il tempo di esecuzione nel caso pessimo utilizzando le euristiche per l'unione per rango e la compressione dei cammini: utilizziamo la foresta di insiemi disgiunti $S = \{S_i\}$, dove ogni insieme S_i (che è esso stesso un albero) corrisponde all'albero T_i nella foresta $\mathcal{F}$. La struttura di albero all'interno dell'insieme S_i, però, non corrisponde necessariamente a quella di T_i: infatti, la realizzazione di S_i non memorizza esattamente la relazione padre-figlio, ma permette nonostante ciò di determinare la profondità di un nodo qualunque in T_i.

L'idea di fondo è di mantenere in ogni nodo v una "pseudodistanza" $d[v]$, che è definita in maniera tale che la somma delle pseudodistanze lungo il cammino da v alla radice del suo insieme S_i corrisponda alla profondità di v in T_i. Ovverosia, se il cammino da v alla sua radice in S_i è dato da $v_0, v_1, \ldots, v_k$, dove $v_0 = v$ e v_k è la radice di S_i, allora la profondità di v in T_i è data da $\sum_{j=0}^{k} d[v_j]$.

b. Si fornisca una realizzazione di MAKE-TREE.

c. Si mostri come va modificata FIND-SET per realizzare FIND-DEPTH. La realizzazione deve eseguire la compressione dei cammini, e il suo tempo di esecuzione deve essere lineare nella lunghezza del cammino di accesso. Si controlli in particolare che tale realizzazione aggiorni le pseudodistanze in maniera corretta.

d. Si mostri come vanno modificate le procedure UNION e LINK per realizzare GRAFT(r, v), che unisce gli insiemi che contengono r e v. Si controlli in particolare che la realizzazione aggiorni le pseudodistanze in maniera corretta. Si noti che la radice di un insieme S_i non è necessariamente la radice dell'albero corrispondente T_i.

e. Si fornisca un limite stretto sul tempo di esecuzione nel caso pessimo di una sequenza di m operazioni MAKE-TREE, FIND-DEPTH e GRAFT, n delle quali sono operazioni MAKE-TREE.

22-3 *L'algoritmo di Tarjan per i minimi antenati comuni off-line*

Il ***minimo antenato comune*** di due nodi u e v in un albero radicato T è il nodo w che è un antenato di entrambi e ha la profondità maggiore in T. Nel ***problema dei minimi antenati comuni off-line*** abbiamo un albero radicato T e un insieme arbitrario $P = \{\{u, v\}\}$ di coppie non ordinate di nodi in T, e vorremmo determinare il minimo antenato comune di ogni coppia in P. Per risolvere il problema dei minimi antenati comuni off-line, la procedura seguente esegue un'ispezione di T con la chiamata di LCA(*root*[T]). Supponiamo che ogni nodo sia colorato WHITE all'inizio dell'esecuzione

```
LCA(u)
1   MAKE-SET(u)
2   ancestor[FIND-SET(u)] ← u
3   for ogni figlio v di u in T
4       do LCA(u)
5          UNION(u, v)
6          ancestor[FIND-SET(u)] ← u
7   color[u] ← BLACK
8   for ogni nodo v tale che {u, v} ∈ P
9       do if color[u] = BLACK
10            then stampa "Il minimo antenato comune di" u "e" v "è" ancestor[FIND-SET(v)]
```

a. Si dimostri che la linea 10 è eseguita esattamente una volta per ogni coppia $\{u, v\} \in P$.

b. Si dimostri che quando viene invocato LCA(u), il numero di insiemi nella struttura di dati per insiemi disgiunti è uguale alla profondità di u in T.

c. Si dimostri che LCA stampa correttamente il minimo antenato comune di u e v per ogni coppia $\{u, v\} \in P$.

d. Si analizzi il tempo di esecuzione di LCA, assumendo che usiamo la realizzazione della struttura di dati per insiemi disgiunti data nel paragrafo 22.3

Note al capitolo

Molti risultati importanti sulle strutture di dati per insiemi disgiunti sono dovuti almeno in parte a R. E. Tarjan. Il limite superiore di $O(m\alpha(m, n))$ fu provato per la prima volta da Tarjan [186, 188], mentre era stato dimostrato in precedenza il limite superiore $O(m \lg n)$ da Hopcroft e Ullman [4, 103]. Tarjan e van Leeuwen [190] discutono delle varianti sull'euristica di compressione dei cammini, includendo "metodi a una passata" che a volte offrono una esecuzione con costanti moltiplicative migliori dei metodi a due passate. Gabow e Tarjan [76] dimostrano che per certe applicazioni le operazioni per insiemi disgiunti possono essere eseguite in tempo $O(m)$.

Tarjan ha mostrato [187] che è necessario un limite inferiore $\Omega(m\alpha(m, n))$ per operazioni su una qualunque struttura di dati per insiemi disgiunti che soddisfi certe condizioni tecniche: questo limite inferiore è stato più tardi generalizzato da Fredman e Saks [74], che hanno mostrato come, nel caso pessimo, si deve accedere a $\Omega(m\alpha(m, n))$ parole di memoria di $\lg n$ bit.

Algoritmi elementari su grafi

23

In questo capitolo verranno presentati diversi metodi per rappresentare e per visitare un grafo. Visitare un grafo significa seguire in modo sistematico gli archi del grafo in modo da visitarne i vertici. Un algoritmo di visita di grafi può scoprire molte proprietà riguardanti la struttura di un grafo: infatti molti algoritmi cominciano con una visita del grafo dato in input per ottenere queste informazioni strutturali. Inoltre, altri algoritmi su grafi sono realizzati come semplici elaborazioni di algoritmi di visita di grafi: quindi le tecniche di visita di grafi costituiscono il nucleo fondamentale del campo degli algoritmi su grafi.

Il paragrafo 23.1 discute le due rappresentazioni computazionali di grafi più comuni: come liste di adiacenza e come matrici di adiacenza. Il paragrafo 23.2 presenta un semplice algoritmo di visita di grafi chiamato visita in ampiezza, e spiega come creare il corrispondente albero BFS (dall'inglese *breadth-first search*). Il paragrafo 23.3 introduce la visita in profondità e mostra alcuni risultati standard sull'ordine in cui la visita in profondità esamina i vertici. Il paragrafo 23.4 presenta una prima applicazione reale della visita in profondità: l'ordinamento topologico di un grafo orientato aciclico. Una seconda applicazione, e cioè la ricerca delle componenti fortemente connesse di un grafo, viene introdotta nel paragrafo 23.5.

23.1 Rappresentazione di grafi

Vi sono due modi standard di rappresentare un grafo $G = (V, E)$: come una collezione di liste di adiacenza o come una matrice di adiacenza. Di solito si preferisce la rappresentazione con liste di adiacenza perché essa fornisce un modo compatto di rappresentare grafi ***sparsi***, cioè quelli per cui $|E|$ è molto minore di $|V|^2$. La maggior parte degli algoritmi su grafi presentati in questo libro assumono che un grafo in input sia rappresentato usando liste di adiacenza. Tuttavia una rappresentazione con matrice di adiacenza può essere preferita quando il grafo è ***denso*** – $|E|$ è vicino a $|V|^2$ – o quando occorre essere in grado di dire rapidamente se vi è un arco che collega due vertici dati. Ad esempio, due degli algoritmi per cammini minimi tra tutte le coppie di vertici presentati nel Capitolo 26 assumono che i loro grafi di input siano rappresentati con matrici di adiacenza.

La ***rappresentazione con liste di adiacenza*** di un grafo $G = (V, E)$ consiste in un vettore *Adj* di $|V|$ liste, una per ogni vertice in V. Per ogni $u \in V$, la lista di adiacenza *Adj*[u] contiene (puntatori a) tutti i vertici v tali che esista un arco $(u, v) \in E$; quindi *Adj*[u] comprende tutti i vertici adiacenti ad u in G. In ogni lista di adiacenza i vertici vengono di solito memorizzati in un ordine arbitrario. La figura 23.1(b) è una rappresentazione con liste di adiacenza del

VI Algoritmi su grafi

Introduzione

I grafi sono strutture di dati molto diffuse in informatica, e di conseguenza gli algoritmi per lavorare con essi sono di fondamentale importanza per questo campo. Vi sono centinaia di problemi computazionali interessanti definiti in termini di grafi: in questa parte ne considereremo solo alcuni tra i più significativi.

Nel Capitolo 23 si vedrà come si può rappresentare un grafo su di un calcolatore e si discuteranno alcuni algoritmi basati sulla visita di un grafo, utilizzando la visita in ampiezza (*breadth-first*) o la visita in profondità (*depth-first*). Si vedranno due applicazioni della visita in profondità: l'ordinamento topologico di un grafo orientato aciclico, e la decomposizione di un grafo orientato nelle sue componenti fortemente connesse.

Nel Capitolo 24 si mostrerà come trovare un albero di copertura minimo per un grafo in cui ogni arco ha associata una lunghezza o "peso", cioè un modo di collegare tutti i vertici del grafo che abbia peso minimo. Gli algoritmi per calcolare gli alberi di copertura minimi sono buoni esempi di algoritmi "greedy" (si veda il Capitolo 17).

Nei Capitoli 25 e 26 si considererà il problema di trovare i cammini minimi tra i vertici di un grafo, assumendo che ogni arco abbia associata una lunghezza o "peso". In particolare il Capitolo 25 sarà dedicato al calcolo dei cammini minimi da un dato vertice sorgente a tutti gli altri vertici, mentre nel Capitolo 26 si considererà il calcolo dei cammini minimi tra tutte le coppie di vertici.

Infine nel Capitolo 27 si mostrerà come calcolare un flusso massimo di materiale in una rete (un grafo orientato) avente una specifica sorgente di materiale, uno specifico pozzo e prefissate capacità per la quantità di materiale che può attraversare ogni singolo arco orientato. Questo problema generale si può presentare in diverse forme, ed un buon algoritmo per il calcolo di flussi massimi può essere usato per risolvere in maniera efficiente una varietà di problemi correlati.

Nella descrizione del tempo di esecuzione di un algoritmo su di un dato grafo $G = (V, E)$, la dimensione dell'input verrà misurata di solito in termini del numero di vertici $|V|$ e del numero di archi $|E|$ del grafo. Quindi vi sono due parametri rilevanti per la descrizione della dimensione dell'input, e non uno solo.

Verrà adottata una comune convenzione notazionale per questi parametri: in notazione asintotica (come la notazione O o la notazione Θ) e *solo* in questa notazione, il simbolo V denota $|V|$ ed il simbolo E denota $|E|$.

Ad esempio, si dirà "l'algoritmo richiede tempo $O(V, E)$", intendendo dire che l'algoritmo richiede tempo $O(|V|, |E|)$. Questa convenzione rende le formule che descrivono il tempo di esecuzione più facili da leggere, senza rischi di ambiguità.

Un'altra convenzione verrà adottata nello pseudocodice: gli insiemi dei vertici e degli archi di un grafo G verranno indicati rispettivamente con $V[G]$ ed $E[G]$. Quindi nello pseudocodice i vertici e gli archi sono visti come attributi di un grafo.

Algoritmi elementari su grafi

23

In questo capitolo verranno presentati diversi metodi per rappresentare e per visitare un grafo. Visitare un grafo significa seguire in modo sistematico gli archi del grafo in modo da visitarne i vertici. Un algoritmo di visita di grafi può scoprire molte proprietà riguardanti la struttura di un grafo: infatti molti algoritmi cominciano con una visita del grafo dato in input per ottenere queste informazioni strutturali. Inoltre, altri algoritmi su grafi sono realizzati come semplici elaborazioni di algoritmi di visita di grafi; quindi le tecniche di visita di grafi costituiscono il nucleo fondamentale del campo degli algoritmi su grafi.

Il paragrafo 23.1 discute le due rappresentazioni computazionali di grafi più comuni: come liste di adiacenza e come matrici di adiacenza. Il paragrafo 23.2 presenta un semplice algoritmo di visita di grafi chiamato visita in ampiezza, e spiega come creare il corrispondente albero BFS (dall'inglese *breadth-first search*). Il paragrafo 23.3 introduce la visita in profondità e mostra alcuni risultati standard sull'ordine in cui la visita in profondità esamina i vertici. Il paragrafo 23.4 presenta una prima applicazione reale della visita in profondità: l'ordinamento topologico di un grafo orientato aciclico. Una seconda applicazione, e cioè la ricerca delle componenti fortemente connesse di un grafo, viene introdotta nel paragrafo 23.5.

23.1 Rappresentazione di grafi

Vi sono due modi standard di rappresentare un grafo $G = (V, E)$: come una collezione di liste di adiacenza o come una matrice di adiacenza. Di solito si preferisce la rappresentazione con liste di adiacenza perché essa fornisce un modo compatto di rappresentare grafi ***sparsi***, cioè quelli per cui $|E|$ è molto minore di $|V|^2$. La maggior parte degli algoritmi su grafi presentati in questo libro assumono che un grafo in input sia rappresentato usando liste di adiacenza. Tuttavia una rappresentazione con matrice di adiacenza può essere preferita quando il grafo è ***denso*** – $|E|$ è vicino a $|V|^2$ – o quando occorre essere in grado di dire rapidamente se vi è un arco che collega due vertici dati. Ad esempio, due degli algoritmi per cammini minimi tra tutte le coppie di vertici presentati nel Capitolo 26 assumono che i loro grafi di input siano rappresentati con matrici di adiacenza.

La ***rappresentazione con liste di adiacenza*** di un grafo $G = (V, E)$ consiste in un vettore Adj di $|V|$ liste, una per ogni vertice in V. Per ogni $u \in V$, la lista di adiacenza $Adj[u]$ contiene (puntatori a) tutti i vertici v tali che esista un arco $(u, v) \in E$; quindi $Adj[u]$ comprende tutti i vertici adiacenti ad u in G. In ogni lista di adiacenza i vertici vengono di solito memorizzati in un ordine arbitrario. La figura 23.1(b) è una rappresentazione con liste di adiacenza del

Figura 23.1 *Due rappresentazioni di un grafo non orientato.* (a) *Un grafo orientato G con cinque vertici e sette archi.* (b) *Una rappresentazione con liste di adiacenza di G.* (c) *La rappresentazione con matrice di adiacenza di G.*

Figura 23.2 *Due rappresentazioni di un grafo orientato.* (a) *Un grafo orientato G con sei vertici ed otto archi.* (b) *Una rappresentazione con liste di adiacenza di G.* (c) *La rappresentazione con matrice di adiacenza di G.*

grafo non orientato di figura 23.1 (a). Analogamente, la figura 23.2(b) è una rappresentazione con liste di adiacenza del grafo orientato di figura 23.2(a).

Se G è un grafo orientato, la somma delle lunghezze di tutte le liste di adiacenza è $|E|$, perché un arco della forma (u, v) è rappresentato ponendo v in $Adj[u]$. Se G è un grafo non orientato, la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$, perché se (u, v) è un arco non orientato, allora u appare nella lista di adiacenza di v e viceversa. Sia per un grafo orientato che per uno non orientato la rappresentazione con liste di adiacenza gode della desiderabile proprietà che la quantità di memoria necessaria è $O(\max(V, E)) = O(V + E)$.

Le liste di adiacenza possono essere facilmente adattate per rappresentare ***grafi pesati***, cioè grafi per i quali ogni arco ha associato un ***peso*** normalmente dato da una ***funzione peso*** $w : E \to \mathbf{R}$. Ad esempio, sia $G = (V, E)$ un grafo pesato con una funzione peso w. Il peso $w(u, v)$ dell'arco $(u, v) \in E$ viene memorizzato semplicemente insieme al vertice v nella lista di adiacenza di u. La rappresentazione con liste di adiacenza è abbastanza robusta, nel senso che può essere adattata a molte altre varianti di grafi.

Uno svantaggio potenziale della rappresentazione con liste di adiacenza è che per determinare se un dato arco (u, v) è presente nel grafo non vi è metodo più veloce che cercare v nella lista di adiacenza $Adj[u]$. Si può porre rimedio a questo svantaggio con una rappresentazione del grafo come matrice di adiacenza, al costo di usare asintoticamente più memoria.

Per la ***rappresentazione con matrice di adiacenza*** di un grafo $G = (V, E)$, si assume che i vertici siano numerati $1, 2, \ldots, |V|$ in modo arbitrario. La rappresentazione con

matrice di adiacenza di un grafo G consiste in una matrice $A = (a_{ij})$ di dimensione $|V| \times |V|$ tale che

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E, \\ 0 & \text{altrimenti}. \end{cases}$$

Le figure 23.1(c) e 23.2(c) sono le matrici di adiacenza rispettivamente del grafo non orientato di figura 23.1(a) e del grafo orientato di figura 23.2(a). La matrice di adiacenza di un grafo richiede memoria $\Theta(V^2)$, indipendentemente dal numero di archi nel grafo.

Si osservi la simmetria rispetto alla diagonale principale della matrice di adiacenza di figura 23.1(c). La ***trasposta*** di una matrice $A = (a_{ij})$ è definita come la matrice $A^T = (a_{ij}^T)$, dove $a_{ij}^T = a_{ji}$. Poiché in un grafo non orientato (u, v) e (v, u) rappresentano lo stesso arco, la matrice di adiacenza di un grafo non orientato è identica alla propria trasposta: $A = A^T$. In alcune applicazioni può essere conveniente memorizzare solo i dati che compaiono sopra la diagonale principale (diagonale inclusa) della matrice di adiacenza, riducendo quindi la memoria necessaria per memorizzare il grafo quasi della metà.

Come per la rappresentazione con liste di adiacenza, anche la rappresentazione con matrice di adiacenza può essere usata per grafi pesati. Ad esempio, se $G = (V, E)$ è un grafo pesato con funzione peso w, il peso $w(u, v)$ dell'arco $(u, v) \in E$ viene memorizzato semplicemente come l'elemento in riga u ed in colonna v della matrice di adiacenza. Se un arco non esiste si può memorizzare un valore NIL nella corrispondente posizione della matrice, anche se per molti problemi può essere conveniente usare un valore come 0 o ∞.

Benché la rappresentazione con liste di adiacenza sia asintoticamente almeno tanto efficiente quanto la rappresentazione con matrici di adiacenza, la semplicità di una matrice di adiacenza può renderla preferibile quando i grafi siano ragionevolmente piccoli. Inoltre, se il grafo non è pesato, vi è un ulteriore vantaggio per la rappresentazione con matrice di adiacenza riguardante la memorizzazione: infatti, invece di usare una intera parola di memoria per ogni elemento della matrice di adiacenza, si può, usare un singolo bit.

Esercizi

23.1-1 Data una rappresentazione con liste di adiacenza di un grafo orientato, quanto tempo ci vuole per calcolare il grado uscente di ogni vertice? Quanto tempo ci vuole per calcolare i gradi entranti?

23.1-2 Si dia una rappresentazione con liste di adiacenza di un albero binario completo con 7 vertici. Si dia una rappresentazione equivalente con matrice di adiacenza. Si assuma che i vertici siano numerati da 1 a 7 come in uno heap binario.

23.1-3 Il ***grafo trasposto*** di un grafo orientato $G = (V, E)$ è il grafo $G^T = (V, E^T)$, dove $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$; quindi G^T è G con tutti gli archi invertiti. Si descrivano degli algoritmi efficienti per calcolare G^T da G, sia per la rappresentazione con liste di adiacenza che per quella con matrice di adiacenza. Si analizzi il tempo di esecuzione degli algoritmi proposti.

23.1-4 Data una rappresentazione con liste di adiacenza di un multigrafo $G = (V, E)$, si descriva un algoritmo di tempo $O(V + E)$ per calcolare la rappresentazione con liste

di adiacenza del grafo non orientato "equivalente" $G' = (V, E')$, dove E' è ottenuto da E sostituendo archi multipli tra due vertici con un unico arco, ed eliminando tutti i cappi.

23.1-5 Il ***quadrato*** di un grafo orientato $G = (V, E)$ è il grafo $G^2 = (V, E^2)$ tale che $(u, w) \in E^2$ se e solo se per un qualche vertice $v \in V$, sia $(u, v) \in E$ che $(v, w) \in E$. Quindi G^2 contiene un arco tra u e w se G contiene un cammino con esattamente due archi tra u e w. Si descrivano degli algoritmi efficienti per calcolare G^2 da G, sia per la rappresentazione con liste di adiacenza che per quella con matrice di adiacenza di G. Si analizzino i tempi di esecuzione degli algoritmi proposti.

23.1-6 Quando si usa una rappresentazione con matrice di adiacenza, la maggior parte degli algoritmi su grafi richiede tempo $\Theta(V^2)$, ma vi sono alcune eccezioni. Si mostri che per determinare se un grafo orientato ha un ***pozzo*** – un vertice con grado entrante $|V| - 1$ e con grado uscente 0 – è sufficiente un tempo $O(V)$, anche se viene utilizzata una matrice di adiacenza.

23.1-7 La ***matrice di incidenza*** di un grafo $G = (V, E)$ è una matrice $B = (b_{ij})$ di dimensione $|V| \times |E|$ tale che

$$b_{ij} = \begin{cases} -1 & \text{se l'arco } j \text{ esce dal vertice } i, \\ 1 & \text{se l'arco } j \text{ entra nel vertice } i, \\ 0 & \text{altrimenti .} \end{cases}$$

Si descriva cosa rappresentano gli elementi della matrice prodotto $B\,B^T$, dove B^T rappresenta la trasposta di B.

23.2 Visita in ampiezza

La ***visita in ampiezza*** (in inglese *breadth-first search*) è uno dei più semplici algoritmi per visitare un grafo e l'archetipo di molti algoritmi importanti su grafi. L'algoritmo di Dijkstra per i cammini minimi con sorgente singola (Capitolo 25) e l'algoritmo di Prim per l'albero di copertura minimo (paragrafo 24.2), ad esempio, usano idee simili a quelle usate nella visita in ampiezza.

Dato un grafo $G = (V, E)$ con uno specifico vertice s chiamato ***sorgente***, la visita in ampiezza esplora sistematicamente gli archi di G per "scoprire" ogni vertice che sia raggiungibile da s. Essa calcola la distanza (cioè il minimo numero di archi) da s ad ognuno dei vertici raggiungibili, e produce un "albero BFS" ("BFS" da *breadth-first search*) che ha s come radice e che comprende tutti i vertici raggiungibili da s. Per ogni vertice v raggiungibile da s, il cammino nell'albero BFS da s a v corrisponde ad un "cammino minimo" da s a v in G, cioè un cammino contenente il minimo numero di archi. L'algoritmo funziona sia per grafi orientati che per grafi non orientati.

La visita in ampiezza è chiamata così perché essa espande la frontiera tra i vertici scoperti e quelli ancora da scoprire in modo uniforme lungo l'ampiezza della frontiera stessa, cioè l'algoritmo scopre tutti i vertici che hanno distanza k da s prima di scoprire un qualunque vertice a distanza $k + 1$.

Per tenere traccia del lavoro in corso, la visita in ampiezza colora ogni vertice di bianco, di grigio o di nero: all'inizio tutti i vertici sono bianchi, e successivamente possono diventare grigi e poi neri. Un vertice viene ***scoperto*** la prima volta che viene incontrato durante la visita: in tale istante esso cessa di essere bianco. Quindi sia i vertici grigi che quelli neri sono già stati scoperti, ma l'algoritmo li distingue per assicurare che la visita proceda appunto in ampiezza. Se $(u, v) \in E$ ed il vertice u è nero, allora il vertice v è grigio o nero; cioè, tutti i vertici adiacenti a vertici neri sono già stati scoperti. Invece i vertici grigi possono avere alcuni vertici adiacenti bianchi; i vertici grigi rappresentano quindi la frontiera tra vertici scoperti e vertici non scoperti.

La visita in ampiezza costruisce un albero BFS che all'inizio contiene solo la radice, che è il vertice sorgente s. Quando un vertice bianco v viene scoperto durante la scansione della lista di adiacenza di un vertice u già scoperto, vengono aggiunti all'albero sia il vertice v che l'arco (u, v): in questo caso si dice che u è il ***padre*** o ***predecessore*** di v nell'albero BFS. Poiché un vertice viene scoperto al massimo una volta, esso ha al massimo un predecessore. Le relazioni di antenato e di discendente nell'albero BFS sono definite relativamente alla radice s nel modo usuale: se u è su un cammino dalla radice s al vertice v, allora u è un antenato di v mentre v è un discendente di u.

```
BFS(G, s)
for ogni vertice u ∈ V[G] − {s}
    do color[u] ← WHITE
       d[u] ← ∞
       π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← {s}
while Q ≠ ∅
    do u ← head[Q]
       for ogni v ∈ Adj[u]
           do if color[v] = WHITE
                 then color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
       DEQUEUE(Q)
       color[u] ← BLACK
```

La procedura di visita in ampiezza che precede, chiamata BFS, assume che il grafo in input $G = (V, E)$ sia rappresentato usando liste di adiacenza. Essa mantiene diverse strutture di dati addizionali associate ad ogni vertice del grafo: ad esempio, il colore di ogni vertice $u \in V$ è memorizzato nella variabile $color[u]$, mentre il predecessore di u viene ricordato nella variabile $\pi[u]$. Se u non ha predecessore (ad esempio se $u = s$ o se u non è stato scoperto), allora

Figura 23.3 *L'effetto dell'algoritmo di visita in ampiezza* BFS *su di un grafo non orientato. Gli archi dell'albero sono mostrati in grigio a mano a mano che vengono prodotti dalla* BFS. *Il valore $d[u]$ è mostrato all'interno di ogni vertice u. Viene mostrato lo stato della coda Q all'inizio di ogni iterazione del ciclo* while *delle linee* 9-18. *Le distanze dei vertici sono mostrate sotto i vertici nella coda.*

$\pi[u]$ = NIL. La distanza dalla sorgente s al vertice u calcolata dall'algoritmo viene memorizzata in $d[u]$. L'algoritmo usa anche una coda FIFO Q (si veda il paragrafo 11.1) per gestire l'insieme dei vertici grigi. La figura 23.3 illustra l'effetto dell'esecuzione della procedura BFS su di uno specifico grafo.

La procedura BFS funziona nel modo seguente. Le linee 1-4 colorano tutti i vertici di bianco, assegnano infinito alla variabile $d[u]$ per ogni vertice u ed inizializzano il predecessore di ogni vertice con NIL. La linea 5 colora il vertice sorgente s di grigio, poiché si assume che esso venga scoperto non appena parte la procedura; la linea 6 inizializza $d[s]$ con 0 e la linea 7 pone NIL come predecessore della sorgente. La linea 8 inizializza Q con la coda che contiene il solo vertice s; da questo momento in poi Q conterrà sempre l'insieme dei vertici grigi.

Il ciclo principale del programma è contenuto nelle linee 9-18. Il ciclo viene ripetuto finché esistono dei vertici grigi, cioè dei vertici già scoperti le cui liste di adiacenza non siano ancora state completamente esaminate. La linea 10 determina il vertice grigio u che si trova in testa alla coda Q. Il ciclo for nelle linee 11-16 esamina ogni vertice v nella lista di adiacenza di u. Se v è bianco (e quindi non è stato ancora scoperto) l'algoritmo lo "scopre" eseguendo le linee 13-16: dapprima esso viene colorato di grigio e la sua distanza $d[v]$ viene posta a $d[u] + 1$; quindi u viene memorizzato come suo predecessore; infine v viene posto in fondo alla coda

Q. Quando tutti i vertici della lista di adiacenza di u sono stati esaminati, nelle linee 17-18 u viene rimosso da Q e viene colorato di nero.

Analisi

Prima di dimostrare le varie proprietà della visita in ampiezza, affrontiamo il problema più semplice di analizzarne il tempo di esecuzione per un grafo di input $G = (V, E)$. Dopo l'inizializzazione nessun vertice verrà mai più colorato di bianco, e quindi il test nella linea 12 assicura che ogni vertice venga inserito nella coda al massimo una volta, e di conseguenza che venga eliminato dalla coda al massimo una volta. Le operazioni di inserimento e di eliminazione dalla coda richiedono tempo $O(1)$, quindi il tempo totale dedicato alle operazioni sulla coda è $O(V)$. Poiché la lista di adiacenza di ogni vertice viene scandita solo quando il vertice è estratto dalla coda, la lista di adiacenza di ogni vertice viene scandita al massimo una volta; inoltre, poiché la somma delle lunghezze di tutte le liste di adiacenza è $\Theta(E)$, il tempo massimo speso per la scansione totale delle liste di adiacenza è $O(E)$. Infine il tempo necessario per l'inizializzazione è $O(V)$, e quindi il tempo totale di esecuzione della procedura BFS è $O(V + E)$: di conseguenza la visita in ampiezza richiede un tempo lineare nella dimensione della rappresentazione con liste di adiacenza di G.

Cammini minimi

All'inizio di questo paragrafo abbiamo affermato che la visita in ampiezza trova la distanza di ogni vertice di un grafo $G = (V, E)$ raggiungibile da un prefissato vertice sorgente $s \in V$. Si definisca la ***distanza sul cammino minimo*** $\delta(s, v)$ da s a v come il minimo numero di archi di un cammino dal vertice s al vertice v, oppure ∞ se non esiste nessun cammino da s a v. Un cammino di lunghezza $\delta(s, v)$ da s a v è chiamato un ***cammino minimo***[1] da s a v. Prima di mostrare che la visita in ampiezza calcola effettivamente le distanze sul cammino minimo, esaminiamo una importante proprietà di queste distanze.

Lemma 23.1

Sia $G = (V, E)$ un grafo orientato o non orientato, e sia $s \in V$ un vertice arbitrario. Allora per ogni arco $(u, v) \in E$

$$\delta(s, v) \le \delta(s, u) + 1 .$$

Dimostrazione. Se u è raggiungibile da s, allora anche v lo è. In questo caso il cammino minimo da s a v non può essere più lungo del cammino minimo da s ad u seguito dell'arco (u, v), e quindi la disuguaglianza è valida. D'altra parte, se u non è raggiungibile da s allora $\delta(s, u) = \infty$; quindi anche in questo caso la disuguaglianza è valida. ■

[1] Nei Capitoli 25 e 26 generalizzeremo lo studio dei cammini minimi al caso dei grafi pesati, in cui ogni arco ha un peso (un numero reale) ed il peso di un cammino è la somma dei pesi dei suoi archi. I grafi considerati nel presente capitolo non sono pesati.

Vogliamo mostrare ora che la procedura BFS calcola correttamente $d[v] = \delta(s, v)$ per ogni vertice $v \in V$. Prima mostriamo che $d[v]$ è un limite superiore per $\delta(s, v)$.

Lemma 23.2

Sia $G = (V, E)$ un grafo orientato o non orientato, e si supponga che la procedura BFS venga eseguita su G a partire da un dato vertice sorgente $s \in V$. Allora, al termine della procedura, per ogni vertice $v \in V$ il valore $d[v]$ calcolato da BFS soddisfa $d[v] \geq \delta(s, v)$.

Dimostrazione. Usiamo un'induzione sul numero di volte che un vertice viene inserito nella coda Q. L'ipotesi induttiva è che $d[v] \geq \delta(s, v)$ per ogni $v \in V$.

La base dell'induzione è la situazione immediatamente dopo che s viene posto in Q, nella linea 8 di BFS. L'ipotesi induttiva è soddisfatta in questo punto, poiché $d[s] = 0 = \delta(s, s)$ e $d[v] = \infty \geq \delta(s, v)$ per ogni $v \in V - \{s\}$.

Per il passo induttivo, si consideri un vertice bianco v che viene scoperto durante la scansione della lista di adiacenza di un vertice u. Poiché l'ipotesi induttiva implica che $d[u] \geq \delta(s, u)$, dall'assegnamento eseguito nella linea 14 e dal lemma 23.1 si ottiene

$$\begin{aligned} d[v] &= d[u] + 1 \\ &\geq \delta(s, u) + 1 \\ &\geq \delta(s, v) . \end{aligned}$$

Il vertice v viene quindi inserito nella coda Q, e non vi sarà inserito mai più nel seguito perché esso viene anche colorato di grigio e la clausola **then** nelle linee 13-16 viene eseguita solo per vertici bianchi. Quindi il valore di $d[v]$ non sarà mai più cambiato e l'ipotesi induttiva continua ad essere soddisfatta. ■

Per dimostrare che $d[v] = \delta(s, v)$ occorre prima mostrare più precisamente come opera la coda Q durante l'esecuzione di BFS. Il prossimo lemma mostra che ad ogni istante vi sono al massimo due distinti valori nella coda.

Lemma 23.3

Si supponga che durante l'esecuzione di BFS su di un grafo $G = (V, E)$ la coda Q contenga i vertici $\langle v_1, v_2, \ldots, v_r \rangle$, dove v_1 è la testa di Q e v_r il fondo. Allora $d[v_r] \leq d[v_1] + 1$ e $d[v_i] \leq d[v_{i+1}]$ per $i = 1, 2, \ldots, r - 1$.

Dimostrazione. La dimostrazione è per induzione sul numero di operazioni sulla coda. All'inizio la coda contiene solo s e quindi l'enunciato del lemma è sicuramente soddisfatto.

Per il passo induttivo, dobbiamo dimostrare che il lemma vale sia dopo aver inserito che dopo aver estratto un vertice dalla coda. Se la testa v_1 della coda viene eliminata, la nuova testa è v_2 (se la coda diventa vuota, allora il lemma ovviamente vale). Ma allora abbiamo che $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, mentre le altre disuguaglianze non vengono modificate: quindi il lemma continua a valere con v_2 come testa. L'inserimento di un vertice nella coda richiede una più attenta analisi del codice. Nella linea 16 di BFS, quando un vertice v viene inserito nella coda diventando v_{r+1}, la testa v_1 di Q è in effetti il vertice u di cui si sta scandendo la lista di adiacenza: quindi $d[v_{r+1}] = d[v] = d[u] + 1 = d[v_1] + 1$. Inoltre abbiamo che $d[v_r] \leq d[v_1] + 1 = d[u] + 1 = d[v] = d[v_{r+1}]$ e le rimanenti disuguaglianze non vengono alterate: quindi il lemma continua a valere quando v viene inserito nella coda. ■

Possiamo ora dimostrare che la visita in ampiezza trova correttamente le distanze sul cammino minimo.

Teorema 23.4 (Correttezza della visita in ampiezza)

Sia $G = (V, E)$ un grafo orientato o non orientato, e si supponga che la procedura BFS venga eseguita su G a partire da un dato vertice sorgente $s \in V$. Allora durante la sua esecuzione BFS scopre ogni vertice $v \in V$ raggiungibile da s, ed al termine si ha $d[v] = \delta(s, v)$ per ogni $v \in V$. Inoltre per ogni vertice $v \neq s$ raggiungibile da s, uno dei cammini minimi da s a v è il cammino minimo da s a $\pi[v]$ seguito dall'arco $(\pi[v], v)$.

Dimostrazione. Consideriamo prima il caso in cui v non sia raggiungibile da s. Poiché per il lemma 23.2 $d[v] \geq \delta(s, v) = \infty$, alla variabile $d[v]$ relativa al vertice v non può essere stato assegnato un valore finito nella linea 14. D'altra parte, con una facile induzione si vede che non ci può essere un primo vertice al cui valore d viene assegnato ∞ nella linea 14. Quindi la linea 14 viene eseguita solo per vertici con un valore d finito, e di conseguenza se v non è raggiungibile esso non sarà mai scoperto.

La parte principale della dimostrazione riguarda i vertici raggiungibili da s. Sia V_k l'insieme dei vertici a distanza k da s: cioè $V_k = \{v \in V : \delta(s, v) = k\}$. La dimostrazione procede per induzione su k. Come ipotesi induttiva, assumiamo che per ogni vertice $v \in V_k$ vi sia esattamente un punto durante l'esecuzione di BFS nel quale

- v viene colorato di grigio,
- k viene assegnato a $d[v]$,
- se $v \neq s$, allora a $\pi[v]$ viene assegnato u per un qualche $u \in V_{k-1}$,
- v viene inserito nella coda Q.

Come abbiamo osservato sopra, vi è sicuramente al più un punto dell'esecuzione che soddisfa queste condizioni.

Il caso base è per $k = 0$: ovviamente abbiamo $V_0 = \{s\}$, poiché la sorgente s è l'unico vertice a distanza 0 da s. Durante l'inizializzazione s è colorato di grigio, $d[s]$ viene posto a 0 ed s viene inserito in Q: quindi l'ipotesi induttiva vale.

Per il passo induttivo, cominciamo con l'osservare che la coda Q non è mai vuota fino al termine dell'algoritmo e che, una volta che un vertice u viene inserito nella coda, né $d[u]$ né $\pi[u]$ vengono più modificati. Quindi, per il lemma 23.3, se i vertici vengono inseriti nella coda durante l'esecuzione dell'algoritmo nell'ordine $v_1, v_2, \ldots, v_r$, allora la sequenza delle distanze è monotona crescente: $d[v_i] \leq d[v_{i+1}]$ per $i = 1, 2, \ldots, r - 1$.

Ora si consideri un arbitrario vertice $v \in V_k$, con $k \geq 1$. La proprietà di monotonicità combinata con $d[v] \geq k$ (per il lemma 23.2) e con l'ipotesi induttiva, implicano che v debba essere scoperto (se mai viene scoperto) solo dopo che tutti i vertici in V_{k-1} siano stati inseriti nella coda.

Poiché $\delta(s, v) = k$, vi è un cammino di k archi da s a v, e quindi deve esistere un vertice $u \in V_k$, tale che $(u, v) \in E$. Senza perdita di generalità, sia u il primo di questi vertici che viene colorato di grigio, il che deve accadere perché per induzione tutti i vertici in V_{k-1} vengono colorati di grigio. Il codice di BFS inserisce nella coda ogni vertice grigio, e quindi u deve comparire prima o poi come testa della coda nella linea 10. Quando u compare come testa della coda, la sua lista di adiacenza viene scandita e il vertice v viene scoperto. Si noti

che il vertice v non può essere stato scoperto prima perché non è adiacente ad alcun vertice in V_j per $j < k - 1$ (altrimenti v non potrebbe appartenere a V_k) e perché, per l'assunzione precedente, u è il primo vertice scoperto in V_{k-1} al quale v è adiacente. La linea 13 rende v grigio, la linea 14 pone $d[v] = d[u] + 1 = k$, la linea 15 assegna u a $\pi[v]$ e la linea 16 inserisce v nella coda. Poiché v è un arbitrario vertice in V_k, l'ipotesi induttiva è dimostrata.

Per concludere la dimostrazione del teorema, si osservi che se $v \in V_k$ allora da ciò che abbiamo appena visto segue che $\pi[v] \in V_{k-1}$; quindi si può ottenere un cammino minimo da s a v prendendo un cammino minimo da s a $\pi[v]$ e poi seguendo l'arco $(\pi[v], v)$. ■

Alberi BFS

La procedura BFS costruisce un albero BFS durante la visita del grafo, come illustrato nella figura 23.3: l'albero è rappresentato dal campo π di ogni vertice. Più formalmente, per un grafo $G = (V, E)$ con sorgente s, si definisce il ***sottografo dei predecessori*** di G come $G_\pi = (V_\pi, E_\pi)$, dove

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

e

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\} .$$

Il sottografo dei predecessori G_π è un ***albero BFS*** se V_π contiene tutti e soli i vertici raggiungibili da s e per ogni $v \in V_\pi$ vi è un unico cammino semplice da s a v in G_π che è anche un cammino minimo da s a v in G. Un albero BFS è effettivamente un albero, poiché è connesso e $|E_\pi| = |V_\pi| - 1$ (si veda il Teorema 5.2). Gli archi in E_π sono chiamati ***archi dell'albero***.

Dopo che la procedura BFS è stata eseguita a partire da un vertice sorgente s su di un grafo G, il lemma seguente mostra che il sottografo dei predecessori è un albero BFS.

Lemma 23.5

Se applicata ad un grafo orientato o non orientato $G = (V, E)$, la procedura BFS costruisce π in modo tale che il sottografo dei predecessori $G_\pi = (V_\pi, E_\pi)$ sia un albero BFS.

Dimostrazione. La linea 15 di BFS assegna u a $\pi[v]$ solo se $(u, v) \in E$ e se $\delta(s, v) < \infty$ (cioè se v è raggiungibile da s), e quindi V_π contiene tutti e soli i vertici in V che siano raggiungibili da s. Poiché G_π forma un albero, esso contiene un unico cammino da s ad ogni vertice in V_π. Applicando il Teorema 23.4 induttivamente, si conclude che ognuno di questi cammini è un cammino minimo. ■

La procedura che segue stampa i vertici di un cammino minimo da s a v, assumendo che BFS sia già stata eseguita per calcolare l'albero dei cammini minimi. Questa procedura richiede un tempo lineare nel numero dei vertici del cammino che viene stampato, poiché ogni chiamata ricorsiva ha come parametro un cammino più corto di un vertice.


```
PRINT-PATH(G, s, v)
if v = s
    then stampa s
    else if π[v] = NIL
            then stampa "non esiste nessun cammino da" s "a" v
            else PRINT-PATH(G, s, π[v])
            stampa v
```

Esercizi

23.2-1 Si mostri il risultato dell'esecuzione della visita in ampiezza sul grafo orientato della figura 23.2(a) usando il vertice 3 come sorgente.

23.2-2 Si mostri il risultato dell'esecuzione della visita in ampiezza sul grafo non orientato della figura 23.3 usando il vertice u come sorgente.

23.2-3 Qual è il tempo di esecuzione della procedura BFS se il grafo di input è rappresentato da una matrice di adiacenza e l'algoritmo viene modificato per gestire questa rappresentazione dell'input?

23.2-4 Si discuta il fatto che, in una visita in ampiezza, il valore $d[u]$ assegnato ad un vertice u è indipendente dall'ordine in cui vengono memorizzati i vertici in ogni lista di adiacenza.

23.2-5 Si dia un esempio di un grafo orientato $G = (V, E)$, un vertice sorgente $s \in V$ ed un insieme di archi dell'albero $E_\pi \subseteq E$ tali che per ogni vertice $v \in V$ l'unico cammino in E_π da s a v è un cammino minimo in G, e tuttavia l'insieme di archi E_π non può essere prodotto dall'esecuzione della procedura BFS su G, indipendentemente dal modo in cui i vertici sono ordinati in ogni lista di adiacenza.

23.2-6 Si dia un algoritmo efficiente per determinare se un grafo non orientato è bipartito.

* ***23.2-7*** Il ***diametro*** di un albero $T = (V, E)$ è dato da

$$\max_{u,v \in V} \delta(u, v) ;$$

cioè il diametro è la più grande di tutte le distanze di cammino minimo nell'albero. Si dia un algoritmo efficiente per calcolare il diametro di un albero, e si analizzi il tempo di esecuzione dell'algoritmo proposto.

23.2-8 Sia $G = (V, E)$ un grafo non orientato. Si dia un algoritmo di tempo $O(V + E)$ per calcolare un cammino in G che attraversi ogni arco in E esattamente una volta in ogni direzione. Si descriva come si può trovare l'uscita da un labirinto se si è provvisti di una grande quantità di monete.

23.3 Visita in profondità

La strategia seguita dalla visita in profondità (in inglese *depth-first search*), come suggerisce il nome stesso, consiste nell'esplorare il grafo andando ad ogni istante il più possibile in "profondità". Nella visita in profondità, gli archi vengono esplorati a partire dall'ultimo vertice scoperto v che abbia ancora degli archi non esplorati uscenti da esso. Quando tutti gli archi di v sono stati esplorati, la visita torna indietro per esplorare gli archi uscenti dal vertice dal quale v era stato scoperto. Questo processo continua finché non vengono scoperti tutti i vertici che sono raggiungibili dal vertice sorgente originario. Se rimane qualche vertice non scoperto, allora uno di essi viene selezionato come nuovo vertice sorgente e la ricerca viene ripetuta a partire da esso: l'intero processo viene ripetuto finché non vengono scoperti tutti i vertici del grafo.

Come nella ricerca in ampiezza, quando un vertice v viene scoperto durante la scansione di una lista di adiacenza di un vertice u già scoperto, la visita in profondità memorizza questo evento assegnando u al campo predecessore $\pi[v]$ di v. A differenza della visita in ampiezza, il cui sottografo dei predecessori forma un albero, il sottografo dei predecessori prodotto da una visita in profondità può essere composto di diversi alberi, perché la visita può essere ripetuta da più sorgenti. Il ***sottografo dei predecessori*** di una visita in profondità è quindi definito in un modo leggermente diverso rispetto a quello di una visita in ampiezza: si pone infatti $G_\pi = (V, E_\pi)$, dove

$$E_\pi = \{(\pi[v], v) : v \in V \quad \text{e} \quad \pi[v] \neq \text{NIL}\} \ .$$

Il sottografo dei predecessori di una visita in profondità forma una ***foresta DFS*** ("DFS" da *depth-first search*) composta di diversi ***alberi DFS***: gli archi in E_π sono chiamati ***archi dell'albero***.

Come per la visita in ampiezza, i vertici vengono colorati durante la visita per indicare il loro stato. Ogni vertice è inizialmente bianco, viene reso grigio quando viene ***scoperto*** durante la visita, e viene colorato di nero quando ne è ***finita la visita***, cioè quando la sua lista di adiacenza è stata completamente esaminata. Questa tecnica garantisce che ogni vertice finisca in esattamente un albero DFS, e quindi che questi alberi siano disgiunti.

Oltre a creare la foresta DFS, la visita in profondità ***marca*** ogni vertice con informazioni temporali. Ogni vertice v ha due etichette: la prima $d[v]$ registra quando v è stato scoperto (e reso grigio), mentre la seconda $f[v]$ registra quando la visita ha finito di esaminare la lista di adiacenza di v (e lo ha reso nero). Queste etichette sono usate in molti algoritmi su grafi e sono utili, in generale, per ragionare sul comportamento della visita in profondità.

La procedura DFS riportata sotto registra nella variabile $d[u]$ quando viene scoperto il vertice u, e nella variabile $f[u]$ quando ne viene finita la visita. Queste etichette sono interi compresi tra 1 e $2|V|$, poiché ognuno dei $|V|$ vertici può essere scoperto una sola volta e la sua visita può finire una sola volta. Per ogni vertice u, si ha

$$d[u] < f[u] \ . \tag{23.1}$$

Ogni vertice u è WHITE prima del tempo $d[u]$, GRAY tra il tempo $d[u]$ ed il tempo $f[u]$ e BLACK nel seguito.

Il seguente pseudocodice è l'algoritmo di base per la visita in profondità. Il grafo in input G può essere orientato o non orientato. La variabile *time* è una variabile globale usata per la marcatura.


```
DFS(G)
1  for ogni vertice u ∈ V[G]
2      do color[u] ← WHITE
3         π[u] ← NIL
4  time ← 0
5  for ogni vertice u ∈ V[G]
6      do if color[u] = WHITE
7            then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY                ▷ Il vertice bianco u è stato appena scoperto
2  d[u] ← time ← time + 1
3  for ogni v ∈ Adj[u]            ▷ Si esplora l'arco (u, v)
4      do if color[v] = WHITE
5            then π[v] ← u
6                 DFS-VISIT(v)
7  color[u] ← BLACK               ▷ Si rende u nero; la sua visita è finita.
8  f[u] ← time ← time + 1
```

La figura 23.4 mostra l'effetto dell'esecuzione della procedura DFS sul grafo mostrato nella figura 23.2.

La procedura DFS funziona nel modo seguente. Le linee 1-3 colorano tutti i vertici di bianco ed inizializzano i loro campi π a NIL; la linea 4 azzera il contatore globale del tempo. Le linee 5-7 controllano tutti i vertici di V, e quando ne trovano uno bianco lo visitano usando la procedura DFS-VISIT. Ogni volta che DFS-VISIT(u) viene invocata nella linea 7, il vertice u diventa la radice di un nuovo albero della foresta DFS. Quando DFS termina, ad ogni vertice u è stato assegnato un tempo di scoperta $d[u]$ ed un tempo di fine visita $f[u]$.

In ogni chiamata DFS-VISIT(u), il vertice u è inizialmente bianco. La linea 1 colora u di grigio e la linea 2 memorizza il tempo di scoperta $d[u]$ incrementando la variabile globale *time* e memorizzandone il valore. Le linee 3-6 esaminano ogni vertice v adiacente a u e visitano ricorsivamente v se esso è bianco. Non appena un vertice $v \in Adj[u]$ viene considerato nella linea 3, diciamo che l'arco (u, v) è stato ***esplorato*** dalla visita in profondità. Infine, dopo che ogni arco uscente da u è stato esplorato, le linee 7-8 colorano u di nero e registrano il tempo di fine visita in $f[u]$.

Qual è il tempo di esecuzione della procedura DFS? I due cicli che si trovano rispettivamente nelle linee 1-3 e 5-7 di DFS richiedono tempo $\Theta(V)$, escluso il tempo necessario per eseguire le chiamate a DFS-VISIT. La procedura DFS-VISIT è chiamata esattamente una volta per ogni vertice $v \in V$, poiché DFS-VISIT viene invocata solo su vertici bianchi e la prima cosa che fa è di colorare il vertice di grigio. Durante una esecuzione di DFS-VISIT(v) il ciclo nelle linee 3-6 viene eseguito $|Adj[v]|$ volte. Poiché

$$\sum_{v \in V} |Adj[v]| = \Theta(E) \ ,$$

Figura 23.4 *L'effetto dell'algoritmo di visita in profondità* DFS *su di un grafo orientato. A mano a mano che gli archi vengono esplorati dall'algoritmo essi vengono mostrati o in grigio (se sono archi dell'albero) o tratteggiati (altrimenti). Archi che non sono dell'albero sono etichettati con* B, C *o* F *a seconda che siano all'indietro, di attraversamento o in avanti. I vertici sono etichettati con una coppia tempo di scoperta/tempo di fine visita.*

il costo totale per l'esecuzione delle linee 3-6 di DFS-VISIT è $\Theta(E)$. Quindi il tempo di esecuzione di DFS è $\Theta(V + E)$.

Proprietà della visita in profondità

La visita in profondità fornisce molte informazioni sulla struttura di un grafo. Forse la proprietà più importante della visita in profondità è che il sottografo dei predecessori G_π forma effettivamente una foresta di alberi, poiché la struttura degli alberi DFS rispecchia fedelmente la struttura delle chiamate ricorsive DFS-VISIT. Più precisamente, si ha che $u = \pi[v]$ se e solo se DFS-VISIT(v) è stata chiamata durante la visita della lista di adiacenza di u.

Un'altra importante proprietà della visita in profondità è che i tempi di scoperta e di fine visita hanno una ***struttura di parentesi***, nel senso che se si rappresentano la scoperta di un vertice u con una parentesi sinistra "(u" e la fine della sua visita con una parentesi destra "u)", allora la storia degli eventi di scoperta e di fine visita di tutti i vertici costituisce una espressione ben formata, nel senso che le parentesi sono correttamente bilanciate. Ad esempio la visita in profondità della figura 23.5(a) corrisponde all'espressione mostrata nella figura 23.5(b). Il teorema che segue enuncia in modo più formale questa proprietà.

Figura 23.5 *Proprietà della visita in profondità.* (a) *Il risultato di una visita in profondità su di un grafo orientato. Le etichette dei vertici e degli archi hanno lo stesso significato che nella figura 23.4.* (b) *Gli intervalli tra il tempo di scoperta ed il tempo di fine visita di ogni vertice corrispondono alla struttura di parentesi mostrata. Ogni rettangolo corrisponde all'intervallo tra il tempo di scoperta e quello di fine visita del corrispondente vertice. Gli archi dell'albero sono mostrati esplicitamente. Se due intervalli hanno una sovrapposizione, allora uno è annidato nell'altro ed il vertice corrispondente all'intervallo più piccolo è un discendente del vertice corrispondente a quello più grande.* (c) *Il grafo della parte (a) ridisegnato con tutti gli archi dell'albero e tutti gli archi in avanti che vanno verso il basso in un albero DFS, e tutti gli archi all'indietro che vanno verso l'alto da un discendente ad un antenato.*

Teorema 23.6 (Teorema delle parentesi)

In ogni visita in profondità di un grafo $G = (V, E)$ (orientato o non orientato), per ogni coppia di vertici u e v, una ed una sola delle seguenti condizioni è soddisfatta:

- gli intervalli $[d[u], f[u]]$ e $[d[v], f[v]]$ sono completamente disgiunti,
- l'intervallo $[d[u], f[u]]$ è contenuto interamente nell'intervallo $[d[v], f[v]]$ e u è un discendente di v nell'albero DFS, oppure
- l'intervallo $[d[v], f[v]]$ è contenuto interamente nell'intervallo $[d[u], f[u]]$ e v è un discendente di u nell'albero DFS.

Dimostrazione. Cominciamo con il caso in cui $d[u] < d[v]$. Vi sono due sottocasi da considerare, a seconda che $d[v] < f[u]$ oppure no. Nel primo sottocaso, poiché $d[v] < f[u]$, allora v era stato scoperto quando u era ancora grigio, e questo implica che v è un discendente di u. Inoltre, poiché v era stato scoperto più recentemente di u, tutti gli archi uscenti da esso vengono esplorati e la visita di v finisce prima che l'algoritmo ritorni ad u e ne finisca la visita. Quindi in questo caso l'intervallo $[d[v], f[v]]$ è interamente contenuto nell'intervallo $[d[u], f[u]]$. Nell'altro sottocaso, poiché vale che $f[u] < d[v]$, abbiamo che la disuguaglianza (23.1) implica che gli intervalli $[d[u], f[u]]$ e $[d[v], f[v]]$ sono disgiunti.

Il caso in cui $d[v] < d[u]$ è del tutto simile, scambiando i ruoli di u e v. ■

Corollario 23.7 (Annidamento degli intervalli dei discendenti)

Un vertice v è un discendente proprio di un vertice u nella foresta DFS di un grafo G (orientato o non orientato) se e solo se $d[u] < d[v] < f[v] < f[u]$.

Dimostrazione. Immediata dal Teorema 23.6. ■

Il prossimo teorema fornisce un'altra caratterizzazione importante di quando un vertice è un discendente di un altro vertice nella foresta DFS.

Teorema 23.8 (Teorema del cammino bianco)

In una foresta DFS di un grafo $G = (V, E)$ (orientato o non orientato), un vertice v è un discendente di un vertice u se e solo se al tempo $d[u]$ in cui la visita scopre u, il vertice v è raggiungibile da u con un cammino contenente esclusivamente vertici bianchi.

Dimostrazione $\Rightarrow$: Si assuma che v sia un discendente di u, e sia w un qualunque vertice su di un cammino tra u e v nell'albero DFS (quindi w è un discendente di u). Per il corollario 23.7 abbiamo che $d[u] < d[w]$, e quindi w è bianco al tempo $d[u]$.

$\Leftarrow$: Si supponga che un vertice v sia raggiungibile da u attraverso un cammino di soli vertici bianchi al tempo $d[u]$, ma che v non divenga un discendente di u nell'albero DFS. Senza perdita di generalità, si assuma che ogni altro vertice lungo il cammino divenga un discendente di u (altrimenti, sia v il vertice più vicino ad u lungo il cammino che non diventi un discendente di u). Sia w il predecessore di v nel cammino: ovviamente w è un discendente di u (w ed u possono anche essere lo stesso vertice) e quindi per il corollario 23.7 si ha che $f[w] \leq f[u]$. Si noti che v deve essere scoperto dopo di u, ma prima che la visita di w finisca, e quindi $d[u] < d[v] < f[w] \leq f[u]$. Dal Teorema 23.6 segue allora che l'intervallo $[d[v], f[v]]$ è contenuto interamente nell'intervallo $[d[u], f[u]]$ e quindi per il corollario 23.7 v deve essere un discendente di u. ■

Classificazione degli archi

Un'altra proprietà interessante della visita in profondità è che la visita può essere usata per classificare gli archi del grafo di input $G = (V, E)$: questa classificazione può essere utilizzata per estrarre importanti informazioni sul grafo. Ad esempio nel prossimo paragrafo vedremo

che un grafo orientato è aciclico se e solo se una visita in profondità non produce archi "all'indietro" (lemma 23.10).

Si possono definire quattro tipi di archi in termini della foresta G_π prodotta da una visita in profondità su G.

1. Gli ***archi dell'albero*** sono gli archi della foresta DFS G_π: un arco (u, v) è un arco dell'albero se v è stato scoperto esplorando l'arco (u, v).
2. Gli ***archi all'indietro*** sono quegli archi (u, v) che connettono un vertice u ad un antenato v in un albero DFS. I cappi vengono considerati come archi all'indietro.
3. Gli ***archi in avanti*** sono quegli archi (u, v) che non sono archi dell'albero e che connettono un vertice u ad un discendente v in un albero DFS.
4. Gli ***archi di attraversamento*** sono tutti gli altri: essi possono connettere vertici nello stesso albero DFS, purché un vertice non sia antenato dell'altro, oppure possono connettere vertici in alberi DFS distinti.

Nelle figure 23.4 e 23.5 gli archi sono etichettati per indicarne il tipo. La figura 23.5(c) mostra anche come si può ridisegnare il grafo di figura 23.5(a) in modo che tutti gli archi dell'albero e gli archi in avanti vadano verso il basso in un albero DFS, mentre tutti gli archi all'indietro vanno verso l'alto: ogni grafo può essere ridisegnato in questo modo.

L'algoritmo DFS può essere modificato in modo da classificare gli archi che incontra. L'idea chiave è che un arco (u, v) può essere classificato sulla base del colore del vertice v che viene raggiunto quando l'arco viene esplorato (solo gli archi in avanti e gli archi di attraversamento non vengono distinti):

1. WHITE indica che l'arco esplorato è un arco dell'albero.
2. GRAY indica che è un arco all'indietro.
3. BLACK indica che si tratta di un arco in avanti o di un arco di attraversamento.

Il primo caso è immediato dalla specifica dell'algoritmo. Per il secondo caso, si osservi che i vertici grigi formano sempre una catena lineare di discendenti che corrisponde alla pila delle invocazioni attive di DFS-VISIT: il numero di vertici grigi è uno più della profondità nella foresta DFS del vertice più recentemente scoperto. L'esplorazione procede sempre a partire dal vertice grigio più profondo, e quindi un arco che raggiunge un altro vertice grigio raggiunge necessariamente un antenato. Il terzo caso gestisce l'ultima possibilità: si può mostrare che un tale arco (u, v) è un arco in avanti se $d[u] < d[v]$ mentre è un arco di attraversamento se $d[u] > d[v]$ (si veda l'Esercizio 23.3-4).

In un grafo non orientato vi può essere qualche ambiguità nella classificazione degli archi, perché (u, v) e (v, u) sono in effetti lo stesso arco. Se allo stesso arco possono essere attribuiti due tipi diversi, la priorità verrà data al tipo che compare per primo nella classificazione presentata sopra. In modo del tutto equivalente, l'arco può essere classificato secondo il tipo attribuito a quello tra (u, v) e (v, u) che viene incontrato per primo durante l'esecuzione dell'algoritmo (si veda l'Esercizio 23.3-5).

Mostriamo ora che in una visita in profondità di un grafo non orientato non compaiono mai archi in avanti o archi di attraversamento.

Teorema 23.9

In una visita in profondità di un grafo non orientato G, ogni arco di G è un arco dell'albero oppure un arco all'indietro.

Dimostrazione. Sia (u, v) un arbitrario arco di G, e si supponga senza perdita di generalità che $d[u] < d[v]$. Allora v deve essere scoperto e la sua visita deve finire prima che finisca la visita di u, perché v è nella lista di adiacenza di u. Se l'arco (u, v) viene esplorato prima nella direzione da u a v, allora (u, v) diventa un arco dell'albero; se invece (u, v) viene esplorato prima nella direzione da v ad u, allora (u, v) è un arco all'indietro perché u è ancora grigio nel momento in cui l'arco viene esplorato per la prima volta. ■

Vedremo molte applicazioni di questi teoremi nei paragrafi seguenti.

Esercizi

23.3-1 Si costruisca una matrice 3 per 3, avente come etichette di righe e di colonne WHITE, GRAY e BLACK. In ogni cella (i, j) si indichi se vi può essere un arco da un vertice di colore i ad un vertice di colore j in qualunque momento durante una visita in profondità di un grafo orientato. Inoltre per ogni arco possibile si indichi di quali tipi può essere. Si costruisca una seconda matrice di questo tipo per la visita in profondità di un grafo non orientato.

23.3-2 Si mostri l'effetto di una visita in profondità sul grafo di figura 23.6. Si assuma che il ciclo **for** delle linee 5-7 della procedura DFS consideri i vertici in ordine alfabetico, e che ogni lista di adiacenza sia ordinata alfabeticamente. Si mostrino i tempi di scoperta e di fine visita di ogni vertice, e la classificazione di ogni arco.

23.3-3 Si mostri la struttura a parentesi della visita in profondità mostrata nella figura 23.4

23.3-4 Si mostri che l'arco (u, v) è

a. un arco dell'albero o un arco in avanti se e solo se $d[u] < d[v] < f[v] < f[u]$,

b. un arco all'indietro se e solo se $d[v] < d[u] < f[u] < f[v]$, e

c. un arco di attraversamento se e solo se $d[v] < f[v] < d[u] < f[u]$.

23.3-5 Si mostri che in un grafo non orientato, classificare un arco (u, v) come arco dell'albero o arco all'indietro a seconda di quale tra (u, v) e (v, u) viene incontrato per primo durante la visita in profondità, è equivalente a classificarlo secondo la priorità dei tipi nello schema di classificazione.

23.3-6 Si dia un controesempio alla congettura che se vi è un cammino da u a v in un grafo orientato G, e se $d[u] < d[v]$ in una visita in profondità di G, allora v è un discendente di u nella foresta DFS prodotta dalla visita.

Figura 23.6 *Il grafo orientato usato negli Esercizi 23.3-2 e 23.5-2.*

23.3-7 Si modifichi lo pseudocodice per la visita in profondità in modo tale che stampi ogni arco del grafo orientato G insieme al suo tipo. Si dica se occorre fare delle modifiche se G non è orientato, ed in tal caso si mostri quali.

23.3-8 Si spieghi come un vertice u di un grafo orientato possa finire in un albero DFS che contiene solo u, anche se u ha sia archi entranti che archi uscenti in G.

23.3-9 Si mostri che una visita in profondità di un grafo non orientato G può essere usata per identificare le componenti connesse di G, e che la foresta DFS contiene tanti alberi quante sono le componenti connesse di G. Più precisamente, si mostri come si deve modificare la visita in profondità in modo che ad ogni vertice v sia assegnata una etichetta intera $cc[v]$ tra 1 e k, dove k è il numero di componenti connesse di G, in modo tale che $cc[u] = cc[v]$ se e solo se u e v sono nella stessa componente connessa.

* *23.3-10* Un grafo orientato $G = (V, E)$ è ***connesso singolarmente*** se $u \rightsquigarrow v$ implica che vi è al massimo un cammino semplice da u a v per tutti i vertici $u, v \in V$. Si dia un algoritmo efficiente per determinare se un grafo orientato è connesso singolarmente oppure no.

23.4 Ordinamento topologico

Questo paragrafo mostra come si possa usare la visita in profondità per effettuare l'ordinamento topologico di grafi orientati aciclici, chiamati anche "dag" dall'inglese *directed acyclic graphs*. Un ***ordinamento topologico*** di un dag $G = (V, E)$ è un ordinamento lineare di tutti i suoi vertici tale che se G contiene un arco (u, v), allora u compare prima di v nell'ordinamento (se il grafo non è aciclico, un ordinamento lineare di questo tipo non è possibile). Un ordinamento topologico di un grafo può essere visto come un ordinamento dei suoi vertici lungo una linea orizzontale in modo che tutti gli archi orientati vadano da sinistra verso destra. Quindi l'ordinamento topologico è qualcosa di diverso dalla usuale nozione di "ordinamento" studiata nella Parte II.

Figura 23.7 (a) *Il Professor Bumstead ordina topologicamente i propri indumenti quando si veste. Ogni arco orientato (u, v) significa che l'indumento u deve essere indossato prima dell'indumento v. I tempi di scoperta e di fine visita di una visita in profondità sono mostrati a fianco di ogni vertice.* (b) *Lo stesso grafo mostrato secondo un ordinamento topologico. I suoi vertici sono posti da sinistra a destra in ordine di tempo di fine visita decrescente. Si noti che tutti gli archi orientati vanno da sinistra verso destra.*

I grafi orientati aciclici sono usati in molte applicazioni per indicare una precedenza tra eventi. La figura 23.7 mostra un esempio che si applica quando il professor Bumstead si veste al mattino: il professore deve indossare alcuni indumenti prima di altri (ad esempio, i calzini prima delle scarpe), mentre altri indumenti possono essere indossati in qualunque ordine (ad esempio, calzini e pantaloni). Un arco orientato (u, v) nel dag di figura 23.7(a) indica che l'indumento u deve essere indossato prima dell'indumento v. Quindi un ordinamento topologico di questo dag fornisce un ordine per vestirsi. La figura 23.7(b) mostra il dag, ordinato topologicamente, come un ordinamento dei vertici lungo una linea orizzontale, in modo tale che tutti gli archi orientati vadano da sinistra verso destra.

Il seguente semplice algoritmo ordina topologicamente un dag.

```
TOPOLOGICAL-SORT(G)
chiama DFS(G) per calcolare i tempi di fine visita f[v] per ogni vertice v
appena la visita di un vertice è finita, inseriscilo in testa ad una lista concatenata
return la lista concatenata dei vertici
```

La figura 23.7(b) mostra come i vertici ordinati topologicamente appaiano in ordine inverso ai loro tempi di visita.

Si può eseguire un ordinamento topologico in tempo $\Theta(V + E)$, poiché la visita in profondità richiede tempo $\Theta(V + E)$ e l'inserimento di ognuno dei $|V|$ vertici in testa alla lista richiede tempo $O(1)$.

Dimostriamo la correttezza di questo algoritmo usando il seguente lemma fondamentale che caratterizza i grafi orientati aciclici.

Lemma 23.10

Un grafo orientato G è aciclico se e solo se una visita in profondità di G non produce archi all'indietro.

Dimostrazione $\Rightarrow$: Si supponga che vi sia un arco all'indietro (u, v). Allora il vertice v è un antenato del vertice u nella foresta DFS; quindi vi è un cammino da v ad u in G, e l'arco all'indietro (u, v) completa il ciclo.

$\Leftarrow$: Si supponga che G contenga un ciclo c: mostriamo allora che una visita in profondità di G produce un arco all'indietro. Sia v il primo vertice di c che viene scoperto, e sia (u, v) l'arco che lo precede in c. Al tempo $d[v]$ vi è un cammino di vertici bianchi da v ad u: per il teorema dei cammini bianchi il vertice u diventerà un discendente di v nella foresta DFS, e quindi (u, v) è necessariamente un arco all'indietro. ■

Teorema 23.11

TOPOLOGICAL-SORT(G) produce un ordinamento topologico di un grafo orientato aciclico G.

Dimostrazione. Si supponga che la procedura DFS sia stata eseguita su di un dato dag $G = (V, E)$ per determinare i tempi di fine visita dei suoi vertici. È sufficiente mostrare che per ogni coppia di vertici distinti $u, v \in V$, se vi è un arco da u a v, allora $f[v] < f[u]$. Si consideri un qualunque arco (u, v) esplorato da DFS(G): quando questo arco viene esplorato, v non può essere grigio, perché altrimenti v sarebbe un antenato di u e (u, v) sarebbe un arco all'indietro, contraddicendo il lemma 23.10. Quindi v deve essere o bianco o nero: se v è bianco, allora v diventerà un discendente di u e quindi $f[v] < f[u]$; se v è nero, allora si ha ugualmente che $f[v] < f[u]$. Di conseguenza per ogni arco (u, v) nel dag abbiamo che $f[v] < f[u]$, e questo dimostra il teorema. ■

Esercizi

23.4-1 Si mostri l'ordine dei vertici che TOPOLOGICAL-SORT produce se eseguito sul dag di figura 23.8, con le assunzioni dell'Esercizio 23.3-2.

23.4-2 Vi sono molti ordinamenti diversi dei vertici di un grafo orientato G che sono ordinamenti topologici di G. TOPOLOGICAL-SORT produce l'ordinamento che è l'inverso di quello indotto dai tempi di fine visita della visita in profondità. Si mostri che non tutti gli ordinamenti topologici possono essere prodotti in questo modo: esiste un grafo G tale che uno degli ordinamenti topologici di G non può essere prodotto da TOPOLOGICAL-SORT, indipendentemente dalla struttura delle liste di adiacenza che viene data per G. Si mostri anche che esiste un grafo per il quale due distinte rappresentazioni con liste di adiacenza producono lo stesso ordinamento topologico.

23.4-3 Si dia un algoritmo che determina se un dato grafo $G = (V, E)$ non orientato contiene un ciclo oppure no. L'algoritmo proposto dovrebbe richiedere tempo $O(V)$, indipendentemente da $|E|$.

Figura 23.8 *Un dag per l'ordinamento topologico.*

23.4-4 Dimostrare o confutare la seguente affermazione: se un grafo orientato G contiene cicli, allora TOPOLOGICAL-SORT(G) produce un ordinamento dei vertici che minimizza il numero di archi "cattivi" che sono incompatibili con l'ordinamento prodotto.

23.4-5 Un altro modo di eseguire un ordinamento topologico su di un grafo orientato aciclico $G = (V, E)$ consiste nel trovare ripetutamente un vertice con grado entrante 0, nel darlo in output e nel rimuoverlo dal grafo insieme a tutti i suoi archi uscenti. Si spieghi come realizzare questa idea in modo che richieda tempo $O(V + E)$. Cosa succede a questo algoritmo se G ha cicli?

23.5 Componenti fortemente connesse

Considereremo ora una classica applicazione della visita in profondità: la scomposizione di un grafo orientato nelle sue componenti fortemente connesse. Questo paragrafo mostra come fare questa decomposizione usando due visite in profondità. Molti algoritmi che operano su grafi orientati cominciano con una scomposizione di questo tipo; questo approccio spesso consente di dividere il problema originale in sottoproblemi, uno per ogni componente fortemente connessa. Successivamente, le soluzioni dei sottoproblemi possono essere combinate seguendo la struttura delle connessioni tra le componenti fortemente connesse: questa struttura può essere rappresentata con un grafo chiamato il grafo "delle componenti", che verrà definito nell'Esercizio 23.5-4.

Si ricordi dal Capitolo 5 che una componente fortemente connessa di un grafo orientato $G = (V, E)$ è un insieme massimale di vertici $U \subseteq V$ tale che per ogni coppia di vertici u e v in U abbiamo sia $u \rightsquigarrow v$ che $v \rightsquigarrow u$, cioè i vertici u e v sono raggiungibili l'uno dall'altro. La figura 23.9 ne mostra un esempio.

Il nostro algoritmo per trovare le componenti fortemente connesse di un grafo $G = (V, E)$ usa il grafo trasposto di G, che è definito nell'Esercizio 23.1-3 come il grafo $G^T = (V, E^T)$, dove $E^T = \{(u, v) : (v, u) \in E\}$; quindi E^T contiene gli archi di G con la direzione rovesciata. Data una rappresentazione con liste di adiacenza di G, il tempo necessario per creare G^T è $O(V + E)$. È interessante notare che G e G^T hanno esattamente le stesse componenti fortemente connesse: u e v sono raggiungibili l'uno dall'altro in G se e solo se sono raggiungibili l'uno dall'altro in G^T. La figura 23.9(b) mostra il grafo trasposto di quello in figura 23.9(a), con le componenti fortemente connesse in grigio.

Figura 23.9 (a) *Un grafo orientato G. Le componenti fortemente connesse di C sono mostrate come regioni grigie. Ogni vertice è etichettato con i suoi tempi di scoperta e di fine visita; gli archi dell'albero sono grigi.* (b) *Il grafo G^T, trasposto di G. L'albero depth-first calcolato nella linea 3 di* STRONGLY-CONNECTED-COMPONENTS *è mostrato con gli archi dell'albero colorati di grigio; ogni componente fortemente connessa corrisponde ad un albero depth-first. I vertici b, c, g ed h (in grigio scuro) sono avi di ogni vertice nella propria componente fortemente connessa; questi vertici sono anche le radici degli alberi depth-first prodotti dalla visita in profondità su G^T.* (c) *Il grafo aciclico delle componenti G^{SCC} ottenuto collassando ogni componente fortemente connessa di G in un singolo vertice.*

Il seguente algoritmo in tempo lineare (per la precisione in tempo $\Theta(V + E)$) calcola le componenti fortemente connesse di un grafo orientato $G = (V, E)$ usando due visite in profondità: una su G ed una su G^T.

STRONGLY-CONNECTED-COMPONENTS(G)

1 chiama DFS(G) per calcolare i tempi di fine visita $f[u]$ per ogni vertice u
2 calcola G^T
3 chiama DFS(G^T), ma nel ciclo principale di DFS, considera i vertici in ordine decrescente di $f[u]$ (come calcolati nella linea 1)
4 dai in output i vertici di ogni albero nella foresta DFS prodotta dal passo 3 come una diversa componente fortemente connessa

Questo algoritmo apparentemente semplice sembra non avere nulla a che fare con le componenti fortemente connesse: nel resto di questo paragrafo svelerò il mistero del suo funzionamento e ne dimostreremo la correttezza. Cominciamo con due utili osservazioni.

Lemma 23.12

Se due vertici sono nella stessa componente fortemente connessa, allora nessun cammino tra di essi può abbandonare la componente fortemente connessa.

Dimostrazione. Siano u e v due vertici nella stessa componente fortemente connessa: per definizione vi sono cammini da u a v e da v ad u. Sia w un vertice su di un qualche cammino $u \rightsquigarrow w \rightsquigarrow v$; quindi w è raggiungibile da u. Inoltre, poiché esiste un cammino $v \rightsquigarrow u$, si ha che u è raggiungibile da w con il cammino $w \rightsquigarrow v \rightsquigarrow u$. Quindi u e w sono nella stessa componente fortemente connessa: poiché w era stato scelto in modo arbitrario, il lemma è dimostrato. ■

Teorema 23.13

In una qualunque visita in profondità, tutti i vertici della stessa componente fortemente connessa sono posti nello stesso albero DFS.

Dimostrazione. Sia r il primo dei vertici di una componente fortemente connessa che viene scoperto: poiché r è il primo, gli altri vertici nella componente fortemente connessa sono bianchi al momento della sua scoperta. Vi sono cammini da r ad ogni altro vertice della componente fortemente connessa, e poiché questi cammini non lasciano mai la componente fortemente connessa (per il lemma 23.12), tutti i vertici su di essi sono bianchi. Quindi, per il teorema del cammino bianco, ogni vertice nella componente fortemente connessa diventa un discendente di r nell'albero DFS. ■

Nel resto di questo paragrafo le notazioni $d[u]$ e $f[u]$ fanno riferimento ai tempi di scoperta e di fine visita calcolati dalla prima visita in profondità nella linea 1 di STRONGLY-CONNECTED-COMPONENTS. Analogamente, la notazione $u \rightsquigarrow v$ indica l'esistenza di un cammino in G, e non in G^T.

Per dimostrare che STRONGLY-CONNECTED-COMPONENTS è corretta, introduciamo la nozione di ***avo*** di un vertice u, denotato con $\phi(u)$: un vertice w è l'avo di u se è raggiungibile da u, e se la sua visita è finita per ultima durante la visita in profondità della linea 1. In altre parole, $\phi(u)$ = quel vertice w tale che $u \rightsquigarrow w$ e $f[w]$ è massimo.

Si noti che $\phi(u) = u$ è possibile perché u è raggiungibile da se stesso, e quindi

$$f[u] \le f[\phi(u)], \tag{23.2}$$

Si può anche mostrare che $\phi(\phi(u)) = \phi(u)$ con il seguente ragionamento. Per ogni coppia di vertici $u, v \in V$,

$$u \rightsquigarrow v \text{ implica } f[\phi(v)] \le f[\phi(u)], \tag{23.3}$$

poiché $\{w : v \rightsquigarrow w\} \subseteq \{w : u \rightsquigarrow w\}$, e l'avo ha il tempo di fine visita massimo tra tutti i vertici raggiungibili. Poiché $\phi(u)$ è raggiungibile da u, la formula (23.3) implica che $f[\phi(\phi(u))] \le f[\phi(u)]$; inoltre dalla disuguaglianza (23.2) segue che $f[\phi(u)] \le f[\phi(\phi(u))]$. Quindi abbiamo che $f[\phi(\phi(u))] = f[\phi(u)]$, e di conseguenza $\phi(\phi(u)) = \phi(u)$, perché due vertici la cui visita finisce nello stesso istante sono in realtà lo stesso vertice.

Come vedremo, ogni componente fortemente connessa ha un vertice che è avo di ogni vertice nella componente stessa; questo avo può essere considerato come un "rappresentante" della componente fortemente connessa. Nella visita in profondità di G, esso è il primo vertice della componente fortemente connessa ad essere scoperto, e la sua visita è l'ultima a finire: nella visita in profondità di G^T, esso è la radice di un albero DFS. Dimostreremo ora queste proprietà.

Il primo teorema giustifica il fatto che $\phi(u)$ viene chiamato un "avo" di u.

Teorema 23.14

In un grafo orientato $G = (V, E)$, l'avo $\phi(u)$ di un qualunque vertice $u \in V$ in una qualunque visita in profondità di G è un antenato di u.

Dimostrazione. Se $\phi(u) = u$, il teorema è ovviamente vero. Se $\phi(u) \ne u$, si considerino i colori dei vertici al tempo $d[u]$: se $\phi(u)$ è nero allora $f[\phi(u)] < f[u]$, il che contraddice la disuguaglianza (23.2); se invece $\phi(u)$ è grigio allora esso è un antenato di u, ed il teorema è dimostrato.

Rimane da dimostrare che $\phi(u)$ non è bianco. Vi sono due casi, a seconda dei colori degli eventuali vertici intermedi sul cammino da u a $\phi(u)$.

1. Se ogni vertice intermedio è bianco, allora $\phi(u)$ diventa un discendente di u per il teorema del cammino bianco; ma allora $f[\phi(u)] < f[u]$, contraddicendo la disuguaglianza (23.2).
2. Se qualche vertice intermedio non è bianco, sia t l'ultimo vertice non bianco sul cammino da u a $\phi(u)$. Allora t deve essere grigio, perché non vi è mai un arco da un vertice nero ad un vertice bianco, ed il successore di t è bianco. Ma allora vi è un cammino di vertici bianchi da t a $\phi(u)$, e quindi $\phi(u)$ è un discendente di t per il teorema del cammino bianco; questo implica che $f[t] > f[\phi(u)]$, contraddicendo la scelta di $\phi(u)$, poiché vi è un cammino da u a t. ■

Corollario 23.15

In ogni visita in profondità di un grafo orientato $G = (V, E)$, per ogni vertice $u \in V$ i vertici u e $\phi(u)$ appartengono alla stessa componente fortemente connessa.

Dimostrazione. Abbiamo che $u \rightsquigarrow \phi(u)$ per la definizione di avo, e $\phi(u) \rightsquigarrow u$ perché $\phi(u)$ è un antenato di u. ■

Il teorema seguente fornisce un risultato più forte sulla relazione tra avi e componenti fortemente connesse.

Teorema 23.16

In un grafo orientato $G = (V, E)$, due vertici $u, v \in V$ appartengono alla stessa componente fortemente connessa se e solo se essi hanno lo stesso avo in una visita in profondità di G.

Dimostrazione $\Rightarrow$: Si assuma che u e v siano nella stessa componente fortemente connessa. Ogni vertice raggiungibile da u è raggiungibile da v e viceversa, poiché vi sono cammini in entrambe le direzioni tra u e v; quindi per la definizione di avo si conclude che $\phi(u) = \phi(v)$.

$\Leftarrow$: Si assuma che $\phi(u) = \phi(v)$. Per il corollario 23.15 u è nella stessa componente fortemente connessa di $\phi(u)$ e v è nella stessa componente fortemente connessa di $\phi(v)$. Quindi u e v sono nella stessa componente fortemente connessa. ■

Con il Teorema 23.16 a disposizione, la struttura dell'algoritmo STRONGLY-CONNECTED-COMPONENTS può essere compresa più facilmente. Le componenti fortemente connesse sono insiemi di vertici con lo stesso avo; inoltre, per il Teorema 23.14 ed il teorema delle parentesi (Teorema 23.6), durante la visita in profondità nella linea 1 di STRONGLY-CONNECTED-COMPONENTS un avo è sia il primo vertice scoperto che il vertice la cui visita finisce per ultima nella sua componente fortemente connessa.

Per capire perché si esegue la visita in profondità nella linea 3 di STRONGLY-CONNECTED-COMPONENTS su G^T, si consideri il vertice r con il massimo tempo di fine visita calcolato dalla visita in profondità nella linea 1. Per la definizione di avo, il vertice r deve essere un avo, perché è il suo proprio avo: esso può raggiungere se stesso e nessun altro vertice nel grafo ha un tempo di fine visita più alto. Quali sono gli altri vertici nella componente fortemente connessa di r? Essi sono quei vertici che hanno r come avo, cioè quelli che possono raggiungere r ma non possono raggiungere nessun nodo che abbia un tempo di fine visita maggiore di $f[r]$. Ma il tempo di fine visita di r è il massimo tra quelli di tutti i vertici in G: quindi la componente fortemente connessa di r contiene semplicemente tutti i vertici che possono raggiungere r. In modo del tutto equivalente, possiamo dire che *la componente fortemente connessa di r contiene tutti e soli i vertici che r può raggiungere in G^T*. Quindi la visita in profondità nella linea 3 identifica tutti i vertici nella componente fortemente connessa di r e li colora di nero (una visita in ampiezza o una qualunque visita dei vertici raggiungibili potrebbe identificare questo insieme altrettanto facilmente).

Dopo che la visita in profondità della linea 3 ha terminato di identificare la componente fortemente connessa di r, essa continua col vertice r' con il più grande tempo di fine visita tra tutti i vertici che non sono nella componente fortemente connessa di r. Il vertice r' deve essere il suo proprio avo poiché non può raggiungere nessun altro vertice con un tempo di fine visita più alto (altrimenti sarebbe stato incluso nella componente fortemente connessa di r). Con un ragionamento simile, ogni vertice che può raggiungere r' e che non è già nero deve essere nella componente fortemente connessa di r'; quindi, a mano a mano che la visita in profondità della linea 3 continua, essa identifica e colora di nero tutti i vertici nella componente fortemente connessa di r', visitando G^T a partire da r'.

Quindi la visita in profondità nella linea 3 estrae le componenti fortemente connesse di G una alla volta: ogni componente è identificata nella linea 7 di DFS da una chiamata a DFS-VISIT con l'avo della componente come argomento. Le chiamate ricorsive all'interno di DFS-VISIT colorano di nero uno dopo l'altro tutti i vertici della componente, e quando DFS-VISIT ritorna a DFS, l'intera componente è stata colorata di nero ed estratta. A questo punto DFS trova il vertice con massimo tempo di fine visita tra quelli che non sono stati resi neri: questo vertice è l'avo di un'altra componente, ed il processo continua.

Il teorema seguente formalizza questo punto.

Teorema 23.17

STRONGLY-CONNECTED-COMPONENTS(G) calcola correttamente le componenti fortemente connesse di un grafo orientato G.

Dimostrazione. Mostriamo per induzione sul numero di alberi DFS trovati durante la visita in profondità di G^T che i vertici di ogni albero formano una componente fortemente connessa. Ogni passo dell'argomento induttivo dimostra che un albero formato durante la visita in profondità di G^T è una componente fortemente connessa, assumendo che tutti gli alberi precedentemente prodotti siano componenti fortemente connesse. La base per l'induzione è banale, poiché per il primo albero prodotto non vi sono alberi prodotti precedentemente, e quindi l'ipotesi è ovviamente vera.

Si consideri ora un albero DFS T con radice r prodotto durante la visita in profondità di G^T. Sia $C(r)$ l'insieme dei vertici aventi r come avo:

$$C(r) = \{v \in V : \phi(v) = r\} .$$

Dimostriamo ora che un vertice u viene posto in T se e solo se $u \in C(r)$.

$\Leftarrow$: Dal Teorema 23.13 segue che ogni vertice in $C(r)$ finisce nello stesso albero DFS. Poiché $r \in C(r)$ ed r è la radice di T, ne segue che ogni elemento di $C(r)$ finisce in T.

$\Leftarrow$: Mostriamo che ogni vertice w tale che $f[\phi(w)] > f[r]$ o $f[\phi(w)] < f[r]$ non viene posto in T, considerando i due casi separatamente. Per induzione sul numero degli alberi trovati, ogni vertice w tale che $f[\phi(w)] > f[r]$ non può essere posto nell'albero T, perché nell'istante in cui r viene scoperto w è già stato posto nell'albero avente $\phi(w)$ come radice. D'altra parte ogni vertice w tale che $f[\phi(w)] < f[r]$ non può venire posto in T, perché questo implicherebbe che w [illegible] r; quindi dalla formula (23.3) e dalla proprietà che $r = \phi(r)$ seguirebbe che $f[\phi(w)] \geq f[\phi(r)] = f[r]$, contraddicendo $f[\phi(w)] < f[r]$.

Quindi T contiene solo quei vertici u per cui $\phi(u) = r$, e di conseguenza T è esattamente uguale alla componente fortemente connessa $C(r)$. Questo completa la dimostrazione induttiva. ■

Esercizi

23.5-1 Come può cambiare il numero di componenti fortemente connesse di un grafo in seguito all'aggiunta di un nuovo arco?

23.5-2 Si mostri come opera la procedura STRONGLY-CONNECTED-COMPONENTS sul grafo della figura 23.6. Più precisamente, si mostrino i tempi di fine visita calcolati nella linea 1 e la foresta prodotta nella linea 3. Si assuma che il ciclo nelle linee 5-7 di DFS consideri i vertici in ordine alfabetico, e che le liste di adiacenza siano ordinate alfabeticamente.

23.5-3 Il professor Deaver sostiene che l'algoritmo per trovare le componenti fortemente connesse può essere semplificato utilizzando il grafo originale (invece del trasposto) nella seconda visita in profondità e scandendo i vertici in ordine di tempo di fine visita *crescente*. Il professore ha ragione?

23.5-4 Si definisca il *grafo delle componenti* di $G = (V, E)$ come il grafo $G^{SCC} = (V^{SCC}, E^{SCC})$, dove V^{SCC} contiene un vertice per ogni componente fortemente connessa di G, e E^{SCC} contiene l'arco (u, v) se vi è un arco orientato da un vertice nella componente fortemente connessa corrispondente ad u ad un vertice nella componente fortemente connessa corrispondente a v. La figura 23.9(c) ne mostra un esempio. Si dimostri che G^{SCC} è un grafo orientato aciclico.

23.5-5 Si dia un algoritmo di tempo $O(V+E)$ per calcolare il grafo delle componenti di un grafo orientato $G = (V, E)$. Si assicuri che nel grafo delle componenti prodotto dall'algoritmo proposto vi sia al massimo un arco tra ogni coppia di vertici.

23.5-6 Dato un grafo orientato $G = (V, E)$, si spieghi come creare un altro grafo $G' = (V, E')$ tale che (a) G' ha le stesse componenti fortemente connesse di G, (b) G' ha lo stesso grafo delle componenti di G, e (c) E' è il più piccolo possibile. Si descriva un algoritmo efficiente per calcolare G'.

23.5-7 Un grafo orientato $G = (V, E)$ è ***semiconnesso*** se per ogni coppia di vertici $u, v \in V$, si ha che $u \rightsquigarrow v$ oppure che $v \rightsquigarrow u$. Si dia un algoritmo efficiente per determinare se G è semiconnesso oppure no. Si dimostri che l'algoritmo proposto è corretto e se ne analizzi il tempo di esecuzione.

Problemi

23-1 *Classificazione degli archi con visita in ampiezza*

Una foresta DFS classifica gli archi di un grafo in archi dell'albero, archi all'indietro, archi in avanti ed archi di attraversamento. Anche un albero BFS può essere usato per classificare gli archi raggiungibili dalla sorgente della visita nelle stesse quattro categorie.

a. Si dimostri che in una visita in ampiezza di un grafo non orientato valgono le seguenti proprietà:

1. Non vi sono né archi all'indietro né archi in avanti.
2. Per ogni arco dell'albero (u, v), si ha che $d[v] = d[u] + 1$.
3. Per ogni arco di attraversamento (u, v), si ha che $d[v] = d[u]$ oppure che $d[v] = d[u] + 1$.

b. Si dimostri che in una visita in ampiezza di un grafo orientato valgono le seguenti proprietà:

1. Non vi sono archi in avanti.
2. Per ogni arco dell'albero (u, v), si ha che $d[v] = d[u] + 1$.
3. Per ogni arco di attraversamento (u, v), si ha che $d[v] \le d[u] + 1$.
4. Per ogni arco all'indietro (u, v), si ha che $0 \le d[v] < d[u]$.

23-2 *Punti di articolazione, ponti e componenti biconnesse*

Sia $G = (V, E)$ un grafo non orientato e connesso. Un ***punto di articolazione*** di G è un vertice la cui rimozione disconnette G. Un ***ponte*** di G è un arco la cui rimozione disconnette G. Una ***componente biconnessa*** di G è un insieme massimale di archi tale che due archi qualunque nell'insieme giacciono su di uno stesso ciclo semplice. La figura 23.10 illustra queste definizioni. Si possono determinare i punti di articolazione, i ponti e le componenti biconnesse usando una visita in profondità. Sia $G_\pi = (V, E_\pi)$ un albero DFS di G.

a. Si dimostri che la radice di G_π è un punto di articolazione di G se e solo se ha almeno due figli in G_π.

Figura 23.10 *I punti di articolazione, i ponti e le componenti biconnesse di un grafo non orientato e connesso, usato nel Problema 23-2. I punti di articolazione sono i vertici in grigio scuro, i ponti sono archi in grigio scuro e le componenti biconnesse sono gli archi nelle regioni grigie, che sono etichettate con una numerazione bcc.*

b. Sia v un vertice in G_π diverso dalla radice. Si dimostri che v è un punto di articolazione di G se e solo se v ha un figlio s tale che non vi è alcun arco all'indietro da s o da un qualunque discendente di s a un antenato proprio di v.

c. Si definisca

$$low[v] = \min \begin{cases} d[v], \\ \{d[w] : (u, w) \text{ è un arco all'indietro per un qualche discendente } u \text{ di } v\}. \end{cases}$$

Si mostri come calcolare $low[v]$ per tutti i vertici $v \in V$ in tempo $O(E)$.

d. Si mostri come calcolare tutti i punti di articolazione di G in tempo $O(E)$.

e. Si dimostri che un arco di G è un ponte se e solo se esso non giace su nessun ciclo semplice di G.

f. Si mostri come calcolare tutti i ponti di G in tempo $O(E)$.

g. Si dimostri che le componenti biconnesse di G costituiscono una partizione dell'insieme contenente gli archi di G che non sono ponti.

h. Si dia un algoritmo di tempo $O(E)$ per etichettare ogni arco e di G con un numero positivo $bcc[e]$ tale che $bcc[e] = bcc[e']$ se e solo se e ed e' sono nella stessa componente biconnessa.

23-3 *Cicli di Eulero*

Un ***ciclo di Eulero*** di un grafo orientato e connesso $G = (V, E)$ è un ciclo che attraversa ogni arco di G esattamente una volta, anche se può visitare un vertice più di una volta.

a. Si mostri che G ha un ciclo di Eulero se e solo se per ogni vertice $v \in V$ vale

$$\text{in-degree}(v) = \text{out-degree}(v).$$

b. Si descriva un algoritmo di tempo $O(E)$ per trovare un ciclo di Eulero di G, se ne esiste almeno uno. (*Suggerimento*: si fondino insieme dei cicli fatti di archi disgiunti).

Note al capitolo

Due eccellenti riferimenti per algoritmi su grafi sono Even [65] e Tarjan [188].

La visita in ampiezza è stata scoperta da Moore [150], nel contesto della ricerca di cammini in labirinti. Indipendentemente Lee [134] ha scoperto lo stesso algoritmo nel contesto di problemi di instradamento di fili elettrici su schede di circuiti.

Hopcroft e Tarjan [102] hanno sostenuto la superiorità della rappresentazione con liste di adiacenza rispetto a quella con matrice di adiacenza per grafi sparsi, e sono stati i primi a riconoscere l'importanza algoritmica della visita in profondità. La visita in profondità è stata ampiamente usata sin dai tardi anni cinquanta, specialmente in programmi di intelligenza artificiale.

Tarjan [185] ha dato un algoritmo lineare per trovare le componenti fortemente connesse. L'algoritmo per trovare le componenti fortemente connesse proposto nel paragrafo 23.5 è stato adattato da Aho, Hopcroft e Ullman [5] che lo attribuiscono a S. R. Kosaraju e M. Sharir. Knuth [121] è stato il primo a dare un algoritmo lineare per l'ordinamento topologico.

Alberi di copertura minimi

24

Nella progettazione di circuiti elettronici è spesso necessario rendere i morsetti di diverse componenti elettricamente equivalenti, collegandoli tra loro. Per interconnettere un insieme di n morsetti si può usare un sistema di $n-1$ fili elettrici, ognuno dei quali collega due morsetti: tra tutti questi sistemi, quello preferibile è di solito quello che usa la quantità minima di filo elettrico.

Si può modellare questo problema di collegamenti elettrici con un grafo non orientato e connesso $G=(V,E)$, dove V è l'insieme dei morsetti, E è l'insieme delle possibili interconnessioni tra coppie di morsetti e per ogni arco $(u,v) \in E$ si ha un peso $w(u,v)$ che specifica il costo (la quantità di filo elettrico necessaria) per connettere u e v. Allora si vuole trovare un sottoinsieme aciclico $T \subseteq E$ che connetta tutti i vertici e tale che venga minimizzato il peso totale

$$w(T) = \sum_{(u,v)\in T} w(u,v).$$

Poiché T è aciclico e collega tutti i vertici, esso deve formare un albero che viene chiamato ***albero di copertura***, poiché esso "ricopre" il grafo G. Il problema di determinare l'albero T viene chiamato il ***problema dell'albero di copertura minimo***.[1] La figura 24.1 mostra un esempio di un grafo connesso con il suo albero di copertura minimo.

In questo capitolo esamineremo due algoritmi per risolvere il problema dell'albero di copertura minimo, e precisamente l'algoritmo di Kruskal e quello di Prim. Entrambi possono essere facilmente realizzati in modo da richiedere tempo $O(E \lg V)$ usando ordinari heap binari; usando invece uno heap di Fibonacci, l'algoritmo di Prim può essere reso più efficiente in modo da richiedere tempo $O(E + V \lg V)$, il che costituisce un miglioramento se $|V|$ è molto più piccolo di $|E|$.

Figura 24.1 *Un albero di copertura minimo per un grafo connesso. Sono indicati i pesi degli archi, e gli archi in grigio costituiscono un albero di copertura minimo. Il peso totale dell'albero mostrato è 37. Questo albero non è unico: rimuovendo l'arco (b,c) e sostituendolo con l'arco (a,h) si ottiene un nuovo albero di copertura con peso 37.*

[1] L'espressione "albero di copertura minimo" è una forma abbreviata dell'espressione "albero di copertura di peso minimo". Infatti non si minimizza, ad esempio, il numero degli archi in T, poiché tutti gli alberi di copertura hanno esattamente $|V| - 1$ archi, per il Teorema 5.2.

I due algoritmi illustrano anche un'euristica di ottimizzazione chiamata strategia "greedy" ("golosa" in italiano). Quando durante l'esecuzione di un algoritmo occorre scegliere una tra varie possibilità, questa strategia prescrive di fare la scelta che è più conveniente al momento. Questa strategia non garantisce in generale di trovare una soluzione globale ottima per qualunque problema, ma per il problema dell'albero di copertura minimo si può dimostrare che alcune strategie greedy producono effettivamente un albero di copertura di peso minimo. Le strategie greedy sono discusse a lungo nel Capitolo 17; anche se questo capitolo può essere letto indipendentemente dal Capitolo 17, i metodi greedy presentati qui sono una classica applicazione delle nozioni teoriche introdotte in quel capitolo.

Il paragrafo 24.1 introduce un algoritmo "generico" per l'albero di copertura minimo che fa crescere progressivamente un albero di copertura aggiungendo un arco alla volta. Il paragrafo 24.2 fornisce due modi di realizzare l'algoritmo generico: il primo, dovuto a Kruskal, è simile all'algoritmo per le componenti connesse del paragrafo 22.1; il secondo, dovuto a Prim, è simile all'algoritmo per il cammino minimo di Dijkstra (paragrafo 25.2).

24.1 Costruzione di un albero di copertura minimo

Si assuma di avere un grafo non orientato e connesso $G = (V, E)$ con una funzione peso $w: E \to \mathbf{R}$ e di voler trovare un albero di copertura minimo per G. I due algoritmi che consideriamo in questo capitolo usano un approccio greedy al problema, ma differiscono nel modo in cui questo approccio viene applicato.

Questa strategia "golosa" è catturata dall'algoritmo "generico" che segue, che fa crescere l'albero di copertura minimo di un arco alla volta. L'algoritmo gestisce un insieme A che è sempre un sottoinsieme di un qualche albero di copertura minimo. Ad ogni passo viene determinato un arco (u, v) che può essere aggiunto ad A senza violare questa proprietà invariante, nel senso che $A \cup \{(u, v)\}$ è ancora un sottoinsieme di un albero di copertura minimo. Un tale arco è chiamato un ***arco sicuro*** per A, perché può essere aggiunto ad A senza distruggere l'invariante.

```
GENERIC-MST(G, w)
1  A ← ∅
2  while A non forma un albero di copertura
3        do   trova un arco (u, v) che sia sicuro per A
4             A ← A ∪ {(u, v)}
5  return A
```

Si noti che dopo la linea 1 l'insieme A soddisfa banalmente l'invariante, cioè esso è un sottoinsieme di un albero di copertura minimo. Il ciclo nelle linee 2-4 mantiene l'invariante, e quindi quando A viene restituito nella linea 5 esso deve essere un albero di copertura minimo. La parte complicata naturalmente sta nel trovare un arco sicuro nella linea 3. Almeno uno ne deve esistere, perché quando la linea 3 viene eseguita l'invariante prescrive che vi sia un albero di copertura T tale che $A \subseteq T$, e se vi è un arco $(u, v) \in T$ tale che $(u, v) \notin A$, allora (u, v) è sicuro per A.

Nel resto di questo paragrafo forniremo una regola (il Teorema 24.1) per riconoscere gli archi sicuri; il prossimo paragrafo descriverà due algoritmi che usano questa regola per trovare gli archi sicuri in modo efficiente.

Figura 24.2 *Due modi di vedere un taglio $(S, V - S)$ del grafo della figura 24.1.* (a) *I vertici nell'insieme S sono mostrati in nero e quelli in $V - S$ sono mostrati in bianco. Gli archi che attraversano il taglio sono quelli che collegano vertici bianchi con vertici neri. L'arco (d, c) è l'unico arco leggero che attraversa il taglio. Un sottoinsieme A degli archi è in grigio; si noti che il taglio $(S, V - S)$ rispetta A, perché nessun arco in A attraversa il taglio.* (b) *Lo stesso grafo con i vertici nell'insieme S a sinistra e quelli in $V - S$ a destra. Un arco attraversa il taglio se collega un vertice a sinistra con un vertice a destra.*

Prima di tutto abbiamo bisogno di alcune definizioni. Un ***taglio*** $(S, V - S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V. La figura 24.2 illustra questa nozione. Si dice che un arco $(u, v) \in E$ ***attraversa*** il taglio $(S, V - S)$ se uno dei suoi estremi è in S e l'altro è in $V - S$; un taglio ***rispetta*** un insieme A di archi se nessun arco di A attraversa il taglio. Un arco è un ***arco leggero*** che attraversa un taglio se il suo peso è minimo tra i pesi degli archi che attraversano quel taglio; si noti che vi può essere più di arco leggero che attraversa un taglio, nel caso vi siano archi con peso uguale. Più in generale, si dice che un arco è un ***arco leggero*** che soddisfa una certa proprietà se il suo peso è minimo tra i pesi degli archi che soddisfano quella proprietà.

La regola per riconoscere gli archi sicuri è data dal seguente teorema.

Teorema 24.1

Sia $G = (V, E)$ un grafo non orientato e connesso con una funzione peso w a valori reali definita su E. Sia A un sottoinsieme di E contenuto in un qualche albero di copertura minimo per G, sia $(S, V - S)$ un qualunque taglio che rispetta A, e sia (u, v) un arco leggero che attraversa $(S, V - S)$. Allora l'arco (u, v) è sicuro per A.

Dimostrazione. Sia T l'albero di copertura minimo che contiene A, e si assuma che T non contenga l'arco leggero (u, v), perché se lo contiene il teorema è dimostrato. Costruiamo allora un altro albero di copertura minimo T' che contiene $A \cup \{(u, v)\}$ usando una tecnica "taglia e incolla", mostrando quindi che (u, v) è un arco sicuro per A.

Figura 24.3 *La dimostrazione del Teorema 24.1. I vertici in S sono neri, ed i vertici in V – S sono bianchi. Non sono mostrati tutti gli archi del grafo G, ma solo quelli che fanno parte dell'albero di copertura minimo T. Gli archi in A sono grigi, e (u, v) è un arco leggero che attraversa il taglio (S, V – S). L'arco (x, y) è un arco sull'unico cammino p da u a v in T. Un albero di copertura minimo T' che contenga (u, v) può essere formato rimuovendo l'arco (x, y) ed aggiungendo (u, v).*

L'arco (u, v) forma un ciclo con gli archi sul cammino p da u a v in T, come mostrato nella figura 24.3. Poiché u e v sono su lati opposti del taglio $(S, V-S)$, vi è almeno un arco in T nel cammino p che attraversa il taglio: sia (x, y) uno di questi archi. L'arco (x, y) non appartiene ad A, perché il taglio rispetta A. Poiché (x, y) è sull'unico cammino da u a v in T, eliminando (x, y) si spezza T in due componenti. Aggiungendo (u, v) si ricollegano le componenti nel formare un nuovo albero di copertura $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$.

Mostriamo ora che T' è un albero di copertura minimo. Poiché (u, v) è un arco leggero che attraversa $(S, V-S)$ e anche (x, y) attraversa questo taglio, $w(u, v) \le w(x, y)$. Quindi

$$\begin{aligned} w(T') &= w(T) - w(x,y) + w(u,v) \\ &\le w(T) . \end{aligned}$$

Ma poiché T è un albero di copertura minimo, $w(T) \le w(T')$, e quindi anche T' deve essere un albero di copertura minimo.

Rimane da dimostrare cmhe (u, v) è effettivamente un arco sicuro per A. Infatti abbiamo che $A \subseteq T'$, poiché $A \subseteq T$ e $(x, y) \notin A$; quindi $A \cup \{(u, v)\} \subseteq T'$. Di conseguenza, poiché T' è un albero di copertura minimo, (u, v) è sicuro per A. ■

Il Teorema 24.1 fornisce una comprensione migliore del funzionamento dell'algoritmo GENERIC-MST su di un grafo connesso $G = (V, E)$. A mano a mano che l'algoritmo procede, l'insieme A è sempre aciclico; altrimenti un albero di copertura minimo che contiene A conterrebbe un ciclo, il che è una contraddizione. In ogni istante dell'esecuzione dell'algoritmo il grafo $G_A = (V, A)$ è una foresta, ed ognuna delle componenti connesse di G_A è un albero. (Alcuni degli alberi possono contenere solo un vertice, come ad esempio quando l'algoritmo comincia: A è vuoto e la foresta contiene $|V|$ alberi, uno per ogni vertice). Inoltre ogni arco sicuro (u, v) per A connette componenti distinte di G_A, poiché $A \cup \{(u, v)\}$ deve essere aciclico.

Il ciclo nelle linee 2-4 di GENERIC-MST viene eseguito $|V| - 1$ volte, poiché i $|V| - 1$ archi di un albero di copertura minimo vengono determinati uno alla volta. Inizialmente, quando $A = \varnothing$, vi sono $|V|$ alberi in G_A, ed ogni iterazione riduce questo numero di 1; quando la foresta contiene un solo albero l'algoritmo termina.

I due algoritmi nel paragrafo 24.2 usano il seguente corollario del Teorema 24.1.

Corollario 24.2

Sia $G = (V, E)$ un grafo non orientato e connesso con una funzione peso w a valori reali definita su E. Sia A un sottoinsieme di E contenuto in un albero di copertura minimo per G, e sia C una componente connessa (un albero) nella foresta $G_A = (V, A)$. Se (u, v) è un arco leggero che connette C a qualche altra componente in G_A, allora (u, v) è sicuro per A.

Dimostrazione. Il taglio $(C, V - C)$ rispetta A: quindi (u, v) è un arco sicuro per questo taglio. ■

Esercizi

24.1-1 Sia (u, v) un arco di peso minimo in un grafo G. Si dimostri che (u, v) appartiene ad un albero di copertura minimo di G.

24.1-2 Il professor Sabatier fa la seguente congettura, che è un inverso del Teorema 24.1. Sia $G = (V, E)$ un grafo non orientato e connesso con una funzione peso w a valori reali definita su E. Sia A un sottoinsieme di E contenuto in un albero di copertura minimo per G, sia $(S, V-S)$ un qualunque taglio di G che rispetti A, e sia (u, v) un arco sicuro per A che attraversi $(S, V-S)$. Allora l'arco (u, v) è un arco leggero per il taglio. Si dimostri che la congettura del professore è sbagliata fornendo un controesempio.

24.1-3 Si mostri che se un arco (u, v) è contenuto in un qualche albero di copertura minimo, allora esso è un arco leggero che attraversa un taglio del grafo.

24.1-4 Si dia un semplice esempio di un grafo tale che l'insieme di tutti gli archi leggeri che attraversano un qualche taglio del grafo non formi un albero di copertura minimo.

24.1-5 Sia e un arco di peso massimo in un ciclo di $G = (V, E)$. Si dimostri che vi è un albero di copertura minimo di $G' = (V, E - \{e\})$ che è anche un albero di copertura minimo di G.

24.1-6 Si mostri che un grafo ha un unico albero di copertura minimo se per ogni taglio del grafo vi è un unico arco leggero che attraversa il taglio. Si mostri che il contrario non è vero fornendo un controesempio.

24.1-7 Si mostri che se tutti i pesi degli archi di un grafo sono positivi, allora ogni sottoinsieme di archi che connette tutti i vertici ed ha un peso totale minimo deve essere un albero. Si dia un esempio per mostrare che questo non vale se si ammettono dei pesi non positivi.

24.1-8 Sia T un albero di copertura minimo di un grafo G, e sia L la lista ordinata dei pesi degli archi di T. Si mostri che per ogni altro albero di copertura minimo T' di G la lista L è anche la lista ordinata dei pesi degli archi di T'.

24.1-9 Sia T un albero di copertura minimo di un grafo $G = (V, E)$, e sia V' un sottoinsieme di V. Sia T' il sottografo di T indotto da V', e sia G' il sottografo di G indotto da V'. Si mostri che se T' è connesso, allora T' è un albero di copertura minimo di G'.

24.2 Gli algoritmi di Kruskal e di Prim

I due algoritmi per trovare un albero di copertura minimo descritti in questo paragrafo sono elaborazioni dell'algoritmo generico. Ognuno di essi usa una specifica regola per determinare un arco sicuro nella linea 3 di GENERIC-MST. Nell'algoritmo di Kruskal l'insieme A è una foresta e l'arco sicuro aggiunto ad A è sempre un arco di peso minimo che connette due componenti distinte. Nell'algoritmo di Prim invece l'insieme A forma un singolo albero e l'arco sicuro che viene aggiunto ad A è sempre un arco di peso minimo che connette l'albero ad un vertice che non appartiene all'albero.

Algoritmo di Kruskal

L'algoritmo di Kruskal è basato direttamente sull'algoritmo per l'albero di copertura minimo presentato nel paragrafo 24.1. Esso individua un arco sicuro da aggiungere alla foresta scegliendo un arco (u, v) di peso minimo tra tutti gli archi che connettono due distinti alberi della foresta. Siano C_1 e C_2 due alberi connessi dall'arco (u, v): siccome (u, v) dev'essere un arco leggero che connette C_1 ad un altro albero, il corollario 24.2 implica che (u, v) è un arco sicuro per C_1. L'algoritmo di Kruskal è un algoritmo greedy perché ad ogni passo aggiunge alla foresta un arco avente il minimo peso possibile.

La nostra realizzazione dell'algoritmo di Kruskal è simile all'algoritmo per calcolare le componenti connesse presentato nel paragrafo 22.1. Esso usa una struttura di dati per insiemi disgiunti per mantenere diversi insiemi disgiunti di elementi. Ogni insieme contiene i vertici di un albero della foresta corrente. L'operazione FIND-SET(u) restituisce un rappresentante dell'insieme che contiene u: quindi si può determinare se due vertici u e v appartengono allo stesso albero verificando se FIND-SET(u) è uguale a FIND-SET(v). La fusione di due alberi viene effettuata dalla procedura UNION.

```
MST-KRUSKAL(G, w)
1  A ← ∅
2  for ogni vertice v ∈ V[G]
3      do MAKE-SET(V)
4  ordina gli archi di E per peso w non decrescente
5  for ogni arco (u, v) ∈ E, in ordine di peso non decrescente,
6      do if FIND-SET(u) ≠ FIND-SET(v)
7          then A ← A ∪ {(u, v)}
8               UNION(u, v)
9  return A
```


L'algoritmo di Kruskal funziona come mostrato nella figura 24.4. Le linee 1-3 inizializzano l'insieme A con l'insieme vuoto e creano $|V|$ alberi, uno per ogni vertice; gli archi in E vengono ordinati in ordine di peso non decrescente nella linea 4. Il ciclo **for** nelle linee 5-8 controlla per ogni arco (u, v) se le estremità u e v appartengono allo stesso albero: se così è, l'arco (u, v) non può essere aggiunto alla foresta senza creare un ciclo e quindi viene scartato. Altrimenti i due vertici appartengono ad alberi diversi e l'arco (u, v) viene aggiunto ad A nella linea 7; nella linea 8 i vertici dei due alberi vengono fusi in un unico insieme.

Il tempo di esecuzione dell'algoritmo di Kruskal per un grafo $G = (V, E)$ dipende dalla realizzazione della struttura di dati per gli insiemi disgiunti. Assumiamo qui che si usi la realizzazione della foresta di insiemi disgiunti presentata nel paragrafo 22.3, con le euristiche di unione per rango e di compressione dei cammini, visto che è la realizzazione asintoticamente più veloce tra quelle conosciute. L'inizializzazione richiede tempo $O(V)$, ed il tempo necessario per ordinare gli archi nella linea 4 è $O(E \lg E)$. Ci sono $O(E)$ operazione sulla foresta di insiemi disgiunti, che in totale richiedono tempo $O(E\,\alpha(E, V))$, dove α è la funzione inversa della funzione di Ackermann definita nel paragrafo 22.4. Poiché $\alpha(E, V) = O(\lg E)$, il tempo di esecuzione totale dell'algoritmo di Kruskal è $O(E \lg E)$.

Algoritmo di Prim

Come l'algoritmo di Kruskal, anche l'algoritmo di Prim è un caso speciale dell'algoritmo generico per trovare un albero di copertura minimo presentato nel paragrafo 24.1. L'algoritmo di Prim opera in modo simile all'algoritmo di Dijkstra per trovare i cammini minimi in un grafo (si veda il paragrafo 25.2). L'algoritmo di Prim ha la proprietà che gli archi nell'insieme A formano sempre un singolo albero. Come illustrato nella figura 24.5, l'albero parte da un arbitrario vertice radice r e cresce finché l'albero non copre tutti i vertici in V. Ad ogni passo viene aggiunto all'albero un arco leggero che collega un vertice in A ad un vertice in $V - A$. Per il corollario 24.2 questa regola aggiunge solo archi che sono sicuri per A, e quindi quando l'algoritmo termina gli archi in A formano un albero di copertura minimo. Questa strategia è greedy perché l'albero viene esteso scegliendo, ad ogni passo, un arco di peso minimo tra quelli possibili.

Il punto fondamentale nel realizzare efficientemente l'algoritmo di Prim è di rendere facile la scelta di un nuovo arco da aggiungere all'albero formato dagli archi in A. Nello pseudocodice presentato più avanti, il grafo connesso G e la radice r dell'albero di copertura minimo da costruire vengono forniti in ingresso all'algoritmo. Durante l'esecuzione dell'algoritmo tutti i vertici che *non* sono nell'albero risiedono in una coda con priorità Q basata su di un campo chiave *key*. Per ogni vertice v, $key[v]$ è il minimo tra i pesi degli archi che collegano v ad un qualunque vertice dell'albero; per convenzione, $key[v] = \infty$ se non esiste nessun arco di questo tipo. Il campo $\pi[v]$ indica il "predecessore" di v nell'albero. Durante l'algoritmo, l'insieme A di GENERIC-MST è mantenuto implicitamente come

$$A = \{(v, \pi[v]) : v \in V - \{r\} - Q\} .$$

Quando l'algoritmo termina, la coda con priorità Q è vuota; l'albero di copertura minimo A per G è quindi

$$A = \{(v, \pi[v]) : v \in V - \{r\}\} .$$

Figura 24.4 *L'esecuzione dell'algoritmo di Kruskal sul grafo della figura 24.1. Gli archi grigi appartengono alla foresta A che viene costruita. Gli archi vengono considerati dall'algoritmo in ordine di peso crescente: in ogni passo dell'algoritmo una freccia punta all'arco che viene considerato in quel momento. Se l'arco unisce due alberi distinti della foresta, esso viene aggiunto alla foresta e i due alberi vengono fusi in uno.*

```
MST-PRIM(G, w, r)
1   Q ← V[G]
2   for   ogni u ∈ Q
3         do key[u] ← ∞
4   key[r] ← 0
5   π[r] ← NIL
6   while Q ≠ ∅
7         do  u ← EXTRACT-MIN(Q)
8             for ogni v ∈ Adj[u]
9                 do if  v ∈ Q e w(u, v) < key[v]
10                       then π[v] ← u
11                            key[v] ← w(u, v)
```

Figura 24.4 (*continua*)

L'algoritmo di Prim funziona come mostrato nella figura 24.5. Le linee 1-4 inizializzano la coda con priorità Q con tutti i vertici, e pongono ∞ nell'attributo *key* di ogni vertice, ad eccezione del vertice r, nel cui attributo *key* viene posto 0. La linea 5 inizializza $\pi[r]$ a NIL, poiché la radice r non ha predecessore. Durante tutto l'algoritmo, l'insieme $V - Q$ contiene i vertici dell'albero che si sta costruendo. La linea 7 identifica un vertice $u \in A$ incidente su di un arco leggero che attraversa il taglio $(V - Q, Q)$ (ad eccezione della prima iterazione, nella quale $u = r$ per la linea 4): eliminando u dall'insieme Q lo si aggiunge all'insieme $V - Q$ dei vertici dell'albero. Le linee 8-11 aggiornano i campi *key* e π di ogni vertice v adiacente a u ma non appartenente all'albero; l'aggiornamento preserva le proprietà invarianti che $key[v] = w(v, \pi[v])$ e che $(v, \pi[v])$ è un arco leggero che collega v a qualche vertice dell'albero.

L'efficienza dell'algoritmo di Prim dipende da come viene realizzata la coda con priorità Q. Se Q viene realizzata come un heap binario (si veda il Capitolo 7), si può usare la procedura BUILD-HEAP per eseguire l'inizializzazione delle linee 1-4 in tempo $O(V)$. Il ciclo viene eseguito $|V|$ volte, e siccome ogni operazione EXTRACT-MIN richiede tempo $O(\lg V)$, il tempo totale per tutte le chiamate di EXTRACT-MIN è $O(V \lg V)$. Il ciclo **for** nelle linee 8-11 viene eseguito in tutto $O(E)$ volte, poiché la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$. All'interno del ciclo **for**, il controllo per l'appartenenza a Q di linea 9 può essere realizzato in tempo costante mantenendo un bit per ogni vertice, che dice se il vertice è in Q oppure no, ed aggiornando il bit quando il vertice viene rimosso da Q. L'assegnamento di linea 11 coinvolge una operazione implicita DECREASE-KEY sullo heap, che può essere realizzata su di uno heap binario in tempo $O(\lg V)$. Quindi il tempo totale richiesto dall'algoritmo di Prim è $O(V \lg V + E \lg V) = O(E \lg V)$, che è asintoticamente uguale alla realizzazione proposta per l'algoritmo di Kruskal.

Tuttavia il tempo di esecuzione asintotico dell'algoritmo di Prim può essere migliorato utilizzando uno heap di Fibonacci. Nel Capitolo 21 abbiamo mostrato che se $|V|$ elementi

Figura 24.5 *L'esecuzione dell'algoritmo di Prim sul grafo della figura 24.1. La radice è a; gli archi grigi appartengono all'albero che viene costruito, ed i vertici nell'albero sono in nero. Ad ogni passo dell'algoritmo, i vertici nell'albero determinano un taglio del grafo, ed un arco leggero che attraversa il taglio viene aggiunto all'albero. Nel secondo passo, ad esempio, l'algoritmo può scegliere se aggiungere l'arco (b, c) oppure l'arco (a, h) all'albero, perché entrambi sono archi leggeri che attraversano il taglio.*

sono organizzati in uno heap di Fibonacci, si può eseguire una operazione EXTRACT-MIN in tempo ammortizzato $O(\lg V)$ ed una operazione DECREASE-KEY (per realizzare la linea 11) in tempo ammortizzato $O(1)$. Quindi se si usa uno heap di Fibonacci per realizzare la coda con priorità Q, il tempo di esecuzione dell'algoritmo di Prim scende a $O(E + V \lg V)$.

Esercizi

24.2-1 L'algoritmo di Kruskal può restituire differenti alberi di copertura per lo stesso grafo di input G, a seconda di come, nella fase di ordinamento, vengono ordinati gli archi aventi peso uguale. Si mostri che per ogni albero di copertura minimo T di G vi è un modo di ordinare gli archi nell'algoritmo di Kruskal che garantisce che il risultato dell'algoritmo sia proprio T.

24.2-2 Si supponga che il grafo $G = (V, E)$ sia rappresentato con una matrice di adiacenza. Si dia una semplice realizzazione dell'algoritmo di Prim in tempo $O(V^2)$ per questo caso.

24.2-3 La realizzazione dell'algoritmo di Prim con uno heap di Fibonacci è asintoticamente più veloce della realizzazione con heap binario per un grafo sparso $G = (V, E)$, in cui $|E| = \Theta(V)$? E per un grafo denso, in cui $|E| = \Theta(V^2)$? In quale relazione devono essere $|E|$ e $|V|$ affinché la realizzazione con uno heap di Fibonacci sia asintoticamente più veloce della realizzazione con heap binario?

24.2-4 Si supponga che tutti i pesi degli archi di un grafo siano interi nell'intervallo da 1 a $|V|$. Quanto efficiente può essere reso l'algoritmo di Kruskal? E quanto efficiente può essere reso se invece i pesi degli archi sono interi nell'intervallo da 1 a W per una qualche costante W?

24.2-5 Si supponga che tutti i pesi degli archi di un grafo siano interi nell'intervallo da 1 a $|V|$. Quanto efficiente può essere reso l'algoritmo di Prim? E quanto efficiente può essere reso se invece i pesi degli archi sono interi nell'intervallo da 1 a W per una qualche costante W?

24.2-6 Si descriva un algoritmo efficiente che, dato un grafo non orientato G, tra tutti gli alberi di copertura di G ne determini uno in cui il massimo tra i pesi degli archi che lo compongono sia minimo.

* *24.2-7* Si supponga che i pesi degli archi di un grafo siano uniformemente distribuiti nell'intervallo semiaperto $[0, 1)$. Quale tra gli algoritmi di Kruskal e di Prim può essere reso più efficiente?

* *24.2-8* Si supponga di aver già calcolato un albero di copertura minimo di un grafo G. Quanto velocemente si può aggiornare l'albero di copertura minimo se viene aggiunto a G un nuovo vertice con i relativi archi incidenti?

Problemi

24-1 *Albero di copertura più piccolo dopo il minimo*

Sia $G = (V, E)$ un grafo non orientato e connesso, con una funzione peso $w: E \to \mathbf{R}$, e si supponga che $|E| > |V|$.

a. Sia T un albero di copertura minimo di G. Si dimostri che esistono due archi $(u, v) \in T$ e $(x, y) \notin T$ tali che $T - \{(u, v)\} \cup T[(x, y)]$ è un albero di copertura più piccolo dopo il minimo di G.

b. Sia T un albero di copertura di G, e per ogni coppia di vertici $u, v \in V$ sia $max[u, v]$ un arco di peso massimo sull'unico cammino tra u e v in T. Si descriva un algoritmo in tempo $O(V^2)$ che, dato T, calcoli $max[u, v]$ per ogni $u, v \in V$.

c. Si dia un algoritmo efficiente per calcolare un albero di copertura più piccolo dopo il minimo per G.

24-2 *Albero di copertura minimo per grafi sparsi*

Per un grafo connesso $G = (V, E)$ molto sparso, si può migliorare il tempo di esecuzione $O(E + V\lg V)$ dell'algoritmo di Prim con heap di Fibonacci con una pre-elaborazione di G avente lo scopo di diminuire il numero di vertici prima che l'algoritmo di Prim venga eseguito. La seguente procedura prende come input un grafo pesato G e restituisce una versione "contratta" di G dopo aver aggiunto alcuni archi all'albero di copertura minimo T in fase di costruzione. Inizialmente, per ogni arco $(u, v) \in E$ assumiamo che $orig[u, v] = (u, v)$ e che $w[u, v]$ sia il peso dell'arco.

```
MST-REDUCE(G, T)
for ogni v ∈ V[G]
    do  mark[v] ← FALSE
        MAKE-SET(v)
for ogni u ∈ V[G]
    do if mark[u] = FALSE
        then  scegli v ∈ Adj[u] tale che w[u, v] sia minimo
              UNION(u, v)
              T ← T ∪ {orig[u, v]}
              mark[u] ← mark[v] ← TRUE
V[G'] ← {FIND-SET(v): v ∈ V[G]}
E[G'] ← ∅
for ogni (x, y) ∈ E[G]
    do  u ← FIND-SET(x)
        v ← FIND-SET(y)
        if (u, v) ∉ E[G']
            then E[G'] ← E[G'] ∪ {(u, v)}
                 orig[u, v] ← orig[x, y]
                 w[u, v] ← w[x, y]
            else if w[x, y] < w[u, v]
                 then orig[u, v] ← orig[x, y]
                      w[u, v] ← w[x, y]
si costruiscano le liste di adiacenza Adj per G'
return G' e T
```


a. Sia T l'insieme di archi restituiti da MST-REDUCE, e sia T' l'albero di copertura minimo del grafo G' restituito dalla procedura. Si dimostri che $T \cup \{orig[x, y]: (x, y) \in T'\}$ è un albero di copertura minimo di G.

b. Si mostri che $|V[G']| < |V|/2$.

c. Si mostri come realizzare MST-REDUCE in modo che richieda tempo $O(E)$ (*Suggerimento*: si usino strutture di dati semplici).

d. Si supponga che vengano eseguite k fasi di MST-REDUCE, usando il grafo prodotto da una fase come input della fase successiva, e accumulando gli archi in T. Si mostri che il tempo di esecuzione complessivo delle k fasi è $O(kE)$.

e. Si supponga che dopo aver eseguito k fasi di MST-REDUCE, si esegua l'algoritmo di Prim sul grafo restituito dall'ultima fase. Si mostri come scegliere k in modo tale che il tempo di esecuzione complessivo sia $O(E\lg\lg V)$. Si deduca che la scelta di k proposta minimizza il tempo di esecuzione asintotico complessivo.

f. Per quali valori di $|E|$ (in funzione di $|V|$) l'algoritmo di Prim con pre-elaborazione è asintoticamente più efficiente dell'algoritmo di Prim senza pre-elaborazione?

Note al capitolo

Tarjan [188] presenta una rassegna sul problema dell'albero di copertura minimo e fornisce dell'eccellente materiale avanzato sul tema. Graham ed Hell [92] hanno scritto una storia del problema dell'albero di copertura minimo.

Tarjan attribuisce il primo algoritmo di albero di copertura minimo ad un articolo del 1926 di O. Boruvka; l'algoritmo di Kruskal è dovuto a Kruskal [131] nel 1956, mentre l'algoritmo conosciuto comunemente come l'algoritmo di Prim è stato effettivamente inventato da Prim [163], ma era anche stato inventato precedentemente da V. Jarník nel 1930.

Il motivo per cui gli algoritmi greedy sono efficaci per la ricerca degli alberi di copertura minimi è che l'insieme delle foreste di un grafo forma un matroide grafico (si veda il paragrafo 17.4).

Il più veloce algoritmo di albero di copertura minimo conosciuto per il caso in cui $|E| = \Omega(V\lg V)$ è l'algoritmo di Prim realizzato con heap di Fibonacci. Per grafi più sparsi, Gabow e altri (*Combinatorica* 6 (1986) pp. 109-122) hanno descritto un algoritmo che richiede tempo $O(E\lg\beta)$, dove $\beta = \beta(|E|, |V|) = \min\{i: \lg^{(i)}|V| \le |E|/|V|\}$.

Cammini minimi con sorgente singola

Un automobilista vuole trovare la strada più corta da Roma a Torino. Avendo a disposizione una carta stradale dell'Italia sulla quale è segnata la distanza tra ogni coppia di incroci adiacenti, come si può determinare questa strada più corta?

Un possibile modo consiste nell'enumerare tutte le strade da Roma a Torino, nel sommare le distanze lungo ognuna delle strade e nello scegliere la più corta. Tuttavia è facile vedere che, anche se non ammettiamo strade che contengono cicli, vi sono milioni di possibilità, la maggior parte delle quali sono completamente inutili da considerare. Ad esempio, la strada da Roma a Torino che passa per Trieste sarebbe chiaramente una scelta poco felice, perché Trieste è qualche centinaio di chilometri fuori strada.

In questo capitolo e nel Capitolo 26 mostreremo come risolvere efficientemente questo problema. In un ***problema di cammini minimi*** viene fornito un grafo orientato e pesato $G = (V, E)$, con una *funzione peso* $w : E \rightarrow \mathbf{R}$ che associa ad ogni arco un peso a valore nei reali. Il ***peso*** di un cammino $p = \langle v_0, v_1, \ldots, v_k \rangle$ è la somma dei pesi degli archi che lo costituiscono:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Il ***peso di cammino minimo*** da u a v è definito come

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{se esiste un cammino da } u \text{ a } v, \\ \infty & \text{altrimenti} \end{cases}$$

Un ***cammino minimo*** dal vertice u al vertice v è definito come un qualunque cammino p con peso $w(p) = \delta(u, v)$.

Nell'esempio precedente Roma-Torino, possiamo modellare la carta stradale come un grafo: i vertici rappresentano gli incroci, gli archi rappresentano segmenti di strade e i pesi degli archi rappresentano le distanze stradali. Lo scopo è di trovare un cammino minimo da un dato incrocio di Roma (ad esempio tra Via dei Fori Imperiali e Via Cavour) ad un dato incrocio di Torino (ad esempio tra Corso Giulio Cesare e Corso Novara).

I pesi degli archi possono essere interpretati come misure diverse da distanze: essi sono spesso usati per rappresentare tempi, costi, penalità, perdite o qualunque altra quantità che si accumula in modo lineare e che si vuole minimizzare.

L'algoritmo di visita in ampiezza del paragrafo 23.2 è un algoritmo di cammino minimo che funziona su grafi non pesati, cioè, grafi in cui ogni arco può essere considerato di peso unitario. Poiché nello studio dei cammini minimi in grafi pesati riemergono molti dei concetti relativi alla visita in ampiezza, si consiglia al lettore di rivedere il paragrafo 23.2 prima di

proseguire.

Varianti

In questo capitolo ci concentreremo sul *problema di cammini minimi con sorgente singola*: dato un grafo $G = (V, E)$, si vuole trovare un cammino minimo da un dato vertice *sorgente* $s \in V$ ad ogni vertice $v \in V$. Molti altri problemi possono essere risolti utilizzando un algoritmo per il problema con sorgente singola, come ad esempio le seguenti varianti.

Problema di cammini minimi con destinazione singola: trovare un cammino minimo da ogni vertice v ad un dato vertice *destinazione* t. Invertendo la direzione di ogni arco, si può ridurre questo problema al problema con sorgente singola.

Problema di cammino minimo tra una coppia: trovare un cammino minimo da u a v dove i vertici u e v sono dati. Se si risolve il problema con sorgente singola rispetto al vertice sorgente u, si risolve anche questo problema. Non si conosce nessun algoritmo per questo problema che sia, nel caso peggiore, asintoticamente più efficiente del migliore algoritmo per sorgente singola.

Problema di cammini minimi tra tutte le coppie: trovare un cammino minimo da u a v per ogni coppia di vertici u e v. Questo problema può essere risolto eseguendo un algoritmo per sorgente singola una volta per ogni vertice; tuttavia, esso può essere risolto di solito in modo più efficiente e la sua struttura è interessante di per sé. Il Capitolo 26 considererà in dettaglio il problema di cammini minimi tra tutte le coppie.

Archi con pesi negativi

In alcune istanze del problema di cammini minimi con sorgente singola, vi possono essere archi i cui pesi sono negativi. Se il grafo $G = (V, E)$ non contiene cicli di peso negativo raggiungibili dalla sorgente s, allora per ogni $v \in V$ il peso di cammino minimo $\delta(s, v)$ rimane ben definito anche se ha un valore negativo. Tuttavia, se vi è un ciclo avente peso negativo raggiungibile da s, i pesi di cammino minimo non sono più ben definiti. Nessun cammino da s ad un vertice nel ciclo può essere un cammino minimo, poiché un cammino di peso minore può sempre essere trovato seguendo il cammino "minimo" proposto e quindi percorrendo il ciclo di peso negativo. Se vi è un ciclo di peso negativo su di un cammino da s a v, si definisce $\delta(s, v) = -\infty$.

La figura 25.1 illustra l'effetto dei pesi negativi sui pesi di cammino minimo. Poiché vi è solo un cammino da s ad a (il cammino $\langle s, a \rangle$), $\delta(s, a) = w(s, a) = 3$; analogamente, vi è solo un cammino da s a b, quindi $\delta(s, b) = w(s, a) + w(a, b) = 3 + (-4) = -1$. Vi sono infiniti cammini da s a c: $\langle s, c \rangle$, $\langle s, c, d, c \rangle$, $\langle s, c, d, c, d, c \rangle$ e così via. Poiché il ciclo $\langle c, d, c \rangle$ ha peso $6 + (-3) = 3 > 0$, il cammino minimo da s a c è $\langle s, c \rangle$ con peso $\delta(s, c) = 5$. Analogamente, il cammino minimo da s a d è $\langle s, c, d \rangle$, con peso $\delta(s, d) = w(s, c) + w(c, d) = 11$. Vi sono poi infiniti cammini da s ad e: $\langle s, e \rangle$, $\langle s, e, f, e \rangle$, $\langle s, e, f, e, f, e \rangle$ e così via. Poiché però il ciclo $\langle e, f, e \rangle$ ha peso $3 + (-6) = -3 < 0$, non vi è un cammino minimo da s ad e: infatti percorrendo il ciclo di peso negativo $\langle e, f, e \rangle$ un numero arbitrario di volte possiamo trovare cammini da s ad e con pesi arbitrariamente piccoli e quindi $\delta(s, e) = -\infty$. In modo analogo, abbiamo che $\delta(s, f) = -\infty$;

Figura 25.1 *Archi a pesi negativi in un grafo orientato. All'interno di ogni vertice è mostrato il suo peso di cammino minimo dalla sorgente s. Poiché i vertici e ed f formano un ciclo di peso negativo raggiungibile da s, essi hanno peso di cammino minimo $-\infty$, e poiché g è raggiungibile da un vertice il cui peso di cammino minimo è $-\infty$, anch'esso ha peso di cammino minimo $-\infty$. Vertici come h, i e j non sono raggiungibili da s e quindi il loro peso di cammino minimo è ∞ anche se giacciono su di un ciclo di peso negativo.*

inoltre, poiché G è raggiungibile da f, possiamo trovare cammini con pesi arbitrariamente piccoli da s a g e quindi anche $\delta(s, g) = -\infty$. I vertici h, i e j formano anch'essi un ciclo di peso negativo, tuttavia essi non sono raggiungibili da s e quindi $\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.

Alcuni algoritmi di cammino minimo, come quello di Dijkstra, assumono che tutti i pesi degli archi nel grafo di input siano non negativi, come nell'esempio della carta stradale. Altri, come l'algoritmo di Bellman-Ford, ammettono pesi negativi nel grafo di input e producono una risposta corretta se non vi sono cicli di peso negativo raggiungibili dalla sorgente. Tipicamente se vi è un ciclo di questo tipo l'algoritmo ne può rilevare e segnalare la presenza.

Rappresentazione dei cammini minimi

Spesso si vogliono calcolare non solo i pesi di cammino minimo, ma anche i vertici su tali cammini. La rappresentazione che usiamo per i cammini minimi è simile a quella usata per gli alberi BFS nel paragrafo 23.2. Dato un grafo $G = (V, E)$, manteniamo per ogni vertice $v \in V$ un *predecessore* $\pi[v]$ che è un altro vertice, oppure NIL. Gli algoritmi di cammino minimo presentati in questo capitolo utilizzano l'attributo π in modo tale che la catena dei predecessori che parte da un vertice v segua, in direzione inversa, un cammino minimo da s a v.

Durante l'esecuzione di un algoritmo di cammino minimo, tuttavia, i valori π non indicano necessariamente i cammini minimi. Come nel caso della visita in ampiezza, saremo interessati al *sottografo dei predecessori* $G_\pi = (V_\pi, E_\pi)$ indotto dai valori π. Ancora una volta definiamo l'insieme dei vertici V_π come l'insieme dei vertici di G con predecessore diverso da NIL, più la sorgente s:

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\} .$$

L'insieme di archi orientati E_π è l'insieme di archi indotto dai valori di π per i vertici in V_π:

Figura 25.2 (a) *Un grafo orientato e pesato con i pesi di cammino minimo dalla sorgente s.* (b) *Gli archi grigi formano un albero di cammini minimi avente come radice la sorgente s.* (c) *Un altro albero di cammini minimi con la stessa radice.*

$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\} .$$

Dimostreremo che i valori π prodotti dagli algoritmi di questo capitolo sono tali che, al termine dell'algoritmo, G_π è un "albero di cammini minimi", cioè, informalmente, un albero con radice che contiene un cammino minimo da una sorgente s ad ogni vertice raggiungibile da s. Un albero di cammini minimi è come un albero BFS del paragrafo 23.2, ma contiene cammini dalla sorgente che sono minimi rispetto ai pesi degli archi invece che rispetto al numero degli archi. Per essere precisi, sia $G = (V, E)$ un grafo orientato e pesato, con funzione peso $w : E \to \mathbf{R}$, e si assuma che G non contenga cicli di peso negativo raggiungibili dal vertice sorgente $s \in V$, cosicché i cammini minimi siano ben definiti. Un ***albero di cammini minimi*** con radice s è un sottografo orientato $G' = (V', E')$, dove $V' \subseteq V$ ed $E' \subseteq E$, tale che

1. V' è l'insieme dei vertici raggiungibili da s in G,
2. G' forma un albero con radice s,
3. per ogni $v \in V'$, l'unico cammino semplice da s a v in G' è un cammino minimo da s a v in G.

I cammini minimi non sono necessariamente unici né lo sono gli alberi di cammini minimi. Per esempio, la figura 25.2 mostra un grafo orientato e pesato e due alberi di cammini minimi con la stessa radice.

Struttura del capitolo

Gli algoritmi di cammino minimo con sorgente singola presentati in questo capitolo sono tutti basati su una tecnica chiamata "rilassamento". Il paragrafo 25.1 inizia con lo studio di alcune importanti proprietà generali dei cammini minimi e quindi mostra alcune importanti proprietà degli algoritmi basati sul rilassamento. L'algoritmo di Dijkstra, che risolve il problema di cammini minimi con sorgente singola quando tutti gli archi hanno pesi non negativi, viene presentato nel paragrafo 25.2. Il paragrafo 25.3 introduce l'algoritmo di Bellman-Ford che viene usato nel caso più generale in cui gli archi possono avere pesi negativi. Se il grafo contiene un ciclo di peso negativo raggiungibile dalla sorgente, l'algoritmo di Bellman-Ford ne rileva la presenza. Il paragrafo 25.4 presenta un algoritmo in tempo lineare per calcolare i cammini minimi da una sorgente singola in grafi orientati aciclici. Infine, il paragrafo 25.5 mostra come l'algoritmo di Bellman-Ford possa essere usato

per risolvere un caso particolare di "programmazione lineare".

La nostra analisi richiede alcune convenzioni per effettuare operazioni aritmetiche con quantità infinite. Assumeremo che per ogni numero reale $a \neq -\infty$, valga $a + \infty = \infty + a = \infty$. Inoltre, affinché le dimostrazioni valgano in presenza di cicli di peso negativo, assumeremo che per ogni numero reale $a \neq -\infty$, valga $a + (-\infty) = (-\infty) + a = -\infty$.

25.1 Cammini minimi e rilassamento

Per capire gli algoritmi di cammino minimo con sorgente singola, è utile comprendere le tecniche che essi usano e le proprietà dei cammini minimi che essi sfruttano. La tecnica principale usata dagli algoritmi presentati in questo capitolo è il rilassamento, cioè un metodo che diminuisce ripetutamente un limite superiore al reale peso di cammino minimo di ogni vertice, finché questo limite superiore non diventa proprio uguale al peso di cammino minimo stesso. In questo paragrafo mostreremo come funziona il rilassamento e dimostreremo formalmente diverse proprietà che esso soddisfa.

Per una prima lettura di questo paragrafo, può essere conveniente saltare le dimostrazioni dei teoremi (leggendo solo gli enunciati) e quindi passare direttamente agli algoritmi presentati nei paragrafi 25.2 e 25.3. Si ponga comunque particolare attenzione al lemma 25.7 che è un risultato chiave per la comprensione degli algoritmi di cammino minimo di questo capitolo. Per una prima lettura si potrebbero anche ignorare completamente i lemmi che riguardano i sottografi dei predecessori e gli alberi di cammini minimi (lemmi 25.8 e 25.9), concentrandosi invece sui lemmi precedenti che riguardano i pesi di cammino minimo.

Sottostruttura ottima di un cammino minimo

Gli algoritmi di cammino minimo normalmente sfruttano la proprietà che un cammino minimo tra due vertici contiene al suo interno altri cammini minimi. Questa proprietà di sottostruttura ottima garantisce l'applicabilità sia della programmazione dinamica (Capitolo 16) che del metodo greedy (Capitolo 17). Infatti, l'algoritmo di Dijkstra è un algoritmo greedy, mentre l'algoritmo di Floyd-Warshall, che trova i cammini minimi tra tutte le coppie di vertici (si veda il Capitolo 26), è un algoritmo di programmazione dinamica. Il lemma ed il corollario seguenti enunciano in modo più preciso la proprietà di sottostruttura ottima dei cammini minimi.

Lemma 25.1 (Sottocammini di cammini minimi sono cammini minimi)

Dato un grafo orientato e pesato $G = (V, E)$ con funzione peso $w : E \to \mathbf{R}$, sia $p = \langle v_1, v_2, \ldots, v_k \rangle$ un cammino minimo dal vertice v_1 al vertice v_k. Per ogni i e j tali che $1 \le i \le j \le k$, sia $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ il sottocammino di p dal vertice v_i al vertice v_j. Allora p_{ij} è un cammino minimo da v_i a v_j.

Dimostrazione. Se si decompone il cammino p come $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$, allora $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Si supponga che esista un cammino p'_{ij} da v_i a v_j con peso $w(p'_{ij}) < w(p_{ij})$. Allora $v_1 \overset{p_{1i}}{\leadsto} v_i \overset{p'_{ij}}{\leadsto} v_j \overset{p_{jk}}{\leadsto} v_k$ sarebbe un cammino da v_1 a v_k con peso $w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$ minore di $w(p)$, il che contraddice l'ipotesi che p sia un cammino minimo da v_1 a v_k. ■

Durante lo studio della visita in ampiezza (paragrafo 23.2), abbiamo dimostrato nel lemma 23.1 una semplice proprietà delle distanze minime in grafi non pesati. Il seguente corollario del lemma 25.1 generalizza la proprietà al caso di grafi pesati.

Corollario 25.2

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w : E \to \mathbf{R}$. Si supponga che un cammino minimo p da una sorgente s ad un vertice v possa essere decomposto in $s \overset{p'}{\leadsto} u \to v$ per un qualche vertice u e cammino p'. Allora il peso di un cammino minimo da s a v è $\delta(s, v) = \delta(s, u) + w(u, v)$.

Dimostrazione. Per il lemma 25.1, il sottocammino p' è un cammino minimo dalla sorgente s al vertice u. Quindi

$$\begin{aligned} \delta(s, v) &= w(p) \\ &= w(p') + w(u, v) \\ &= \delta(s, u) + w(u, v) . \end{aligned}$$

■

Il lemma successivo fornisce una semplice ma utile proprietà dei pesi di cammino minimo.

Lemma 25.3

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w : E \to \mathbf{R}$ e vertice sorgente s. Allora per ogni arco $(u, v) \in E$ vale $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Dimostrazione. Un cammino minimo p dalla sorgente s al vertice v non può avere peso maggiore di un qualunque altro cammino da s a v. In particolare, il cammino p non può avere peso maggiore del cammino formato da un cammino minimo dalla sorgente s al vertice u e dall'arco (u, v).

Rilassamento

Gli algoritmi presentati in questo capitolo usano la tecnica del ***rilassamento***. Per ogni vertice $v \in V$, si mantiene un attributo $d[v]$ che è un limite superiore al peso di un cammino minimo dalla sorgente s a v: chiameremo $d[v]$ una ***stima di cammino minimo***. Le stime di cammino minimo e i predecessori vengono inizializzati dalla seguente procedura.

```
INITIALIZE-SINGLE-SOURCE(G, s)
for ogni vertice v ∈ V[G]
        do   d[v] ← ∞
             π[v] ← NIL
d[s] ← 0
```

Dopo l'inizializzazione, $\pi[v] = \text{NIL}$ per tutti i $v \in V$, $d[v] = 0$ per $v = s$, e $d[v] = \infty$ per $v \in V - \{s\}$.

Figura 25.3 *Rilassamento di un arco (u, v) con peso $w(u, v) = 2$. All'interno di ogni vertice è mostrata la stima di cammino minimo.* (a) *Poiché $d[v] > d[u] + w(u, v)$ prima del rilassamento, il valore di $d[v]$ decresce.* (b) *Qui prima del rilassamento abbiamo che $d[v] \le d[u] + w(u, v)$, quindi $d[v]$ non viene modificato.*

Il processo di ***rilassare***[1] un arco (u, v) consiste nel verificare se si può migliorare il cammino minimo per v trovato fino a quel momento passando per u e, in questo caso, nell'aggiornare $d[v]$ e $\pi[v]$. Un passo di rilassamento può diminuire il valore della stima di cammino minimo $d[v]$ e può aggiornare il campo predecessore di v, $\pi[v]$. Il codice che segue effettua un passo di rilassamento sull'arco (u, v).

```
RELAX(u, v, w)
if d[v] > d[u] + w(u, v)
        then   d[v] ← d[u] + w(u, v)
               π[v] ← u
```

La figura 25.3 mostra due esempi di rilassamento di un arco, uno nel quale la stima di cammino minimo decresce ed uno in cui non viene alterata alcuna stima.

Tutti gli algoritmi presentati in questo capitolo invocano INITIALIZE-SINGLE-SOURCE, quindi rilassano ripetutamente gli archi; inoltre, il rilassamento è l'unico mezzo con cui le stime di cammino minimo ed i predecessori cambiano. Gli algoritmi in questo capitolo differiscono per il numero di volte in cui ogni arco viene rilassato e per l'ordine in cui si rilassano gli archi. Nell'algoritmo di Dijkstra e nell'algoritmo di cammino minimo per grafi orientati aciclici, ogni arco viene rilassato esattamente una volta, mentre nell'algoritmo di Bellman-Ford ogni arco viene rilassato più volte.

Proprietà del rilassamento

La correttezza degli algoritmi presentati in questo capitolo dipende da alcune proprietà importanti del rilassamento che sono riassunte nei prossimi lemma. La maggior parte di questi lemma descrive il risultato dell'esecuzione di una sequenza di passi di rilassamento sugli archi di un grafo orientato e pesato, inizializzato con INITIALIZE-SINGLE-SOURCE.

[1] Può sembrare strano che il termine "rilassamento" venga usato per un'operazione che abbassa un limite superiore. L'uso del termine ha origini storiche. Il risultato di un passo di rilassamento può essere visto come un rilassamento del vincolo $d[v] \le d[u] + w(u, v)$ che, per il lemma 25.3, deve essere soddisfatto se $d[u] = \delta(s, u)$ e $d[v] = \delta(s, v)$; cioè, se $d[v] \le d[u] + w(u, v)$, allora non vi è "pressione" per soddisfare questo vincolo e quindi il vincolo è "rilassato".

Ad eccezione del lemma 25.9, questi risultati si applicano ad una *qualunque* sequenza di passi di rilassamento, e non solo a quelli che producono valori di cammino minimo.

Lemma 25.4

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w: E \to \mathbf{R}$, e sia $(u, v) \in E$. Allora, immediatamente dopo il rilassamento dell'arco (u, v) tramite RELAX(u, v, w), si ha che $d[v] \le d[u] + w(u, v)$.

Dimostrazione. Se subito prima di rilassare l'arco (u, v) si aveva $d[v] > d[u] + w(u, v)$, allora subito dopo si ha $d[v] = d[u] + w(u, v)$. Se invece si aveva $d[v] \le d[u] + w(u, v)$, allora né $d[v]$ né $d[u]$ cambiano, quindi $d[v] \le d[u] + w(u, v)$ continua a valere. ■

Lemma 25.5

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w : E \to \mathbf{R}$. Sia $s \in V$ un vertice sorgente e si supponga che il grafo sia inizializzato con INITIALIZE-SINGLE-SOURCE(G, s). Allora $d[v] \ge \delta(s, v)$, per ogni $v \in V$, e questa proprietà invariante viene mantenuta da qualunque sequenza di passi di rilassamento sugli archi di G. Inoltre, una volta che $d[v]$ raggiunge il limite inferiore $\delta(s, v)$, esso non cambia più.

Dimostrazione. La proprietà invariante $d[v] \ge \delta(s, v)$ è sicuramente vera dopo l'inizializzazione, poiché $d[s] = 0 \ge \delta(s, s)$ (si noti che $\delta(s, s)$ vale $-\infty$ se s è in un ciclo di peso negativo, 0 altrimenti), mentre $d[v] = \infty$ implica $d[v] \ge \delta(s, v)$ per ogni $v \in V - \{s\}$. Dimostreremo per assurdo che l'invariante viene mantenuto da una qualunque sequenza di passi di rilassamento. Sia v il primo vertice per cui un passo di rilassamento causa $d[v] < \delta(s, v)$. Allora, subito dopo aver rilassato l'arco (u, v), si ha

$$\begin{aligned} d[u] + w(u, v) &= d[v] \\ &< \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \qquad \text{(per il lemma 25.3)} , \end{aligned}$$

che implica $d[u] < \delta(s, u)$. Ma poiché il rilassamento dell'arco (u, v) non modifica $d[u]$, questa disuguaglianza doveva essere vera subito prima del rilassamento dell'arco, il che contraddice la scelta di v come il primo vertice per il quale $d[v] < \delta(s, v)$. Di conseguenza, l'invariante $d[v] \ge \delta(s, v)$ deve essere mantenuto per tutti i $v \in V$.

Per vedere che il valore di $d[v]$ non muta allorché $d[v] = \delta(s, v)$, è sufficiente notare che, una volta raggiunto il suo limite inferiore, $d[v]$ non può decrescere perché abbiamo appena mostrato che $d[v] \ge \delta(s, v)$ e non può aumentare perché i passi di rilassamento non aumentano i valori dell'attributo d. ■

Corollario 25.6

Si supponga che in un grafo orientato e pesato $G = (V, E)$ con funzione peso $w : E \to \mathbf{R}$ non vi sia nessun cammino che colleghi un vertice sorgente $s \in V$ ad un dato vertice $v \in V$. Allora, dopo che il grafo è stato inizializzato con INITIALIZE-SINGLE-SOURCE(G, s), si ha che $d[v] = \delta(s, v)$ e questa uguaglianza viene mantenuta invariante in qualunque sequenza di passi di rilassamento sugli archi di G.

Dimostrazione. Per il lemma 25.5, si ha che $\infty = \delta(s, v) \le d[v]$ quindi, necessariamente, $d[v] = \infty = \delta(s, v)$.

Il prossimo lemma è cruciale per dimostrare la correttezza degli algoritmi di cammino minimo che appaiono più avanti in questo capitolo: esso fornisce delle condizioni sufficienti per garantire che il rilassamento faccia convergere le stime di cammino minimo ai pesi di cammino minimo.

Lemma 25.7

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w : E \to \mathbf{R}$, sia $s \in V$ un vertice sorgente e sia $s \rightsquigarrow u \to v$ un cammino minimo in G, con $u, v \in V$. Si supponga che G sia inizializzato con INITIALIZE-SINGLE-SOURCE(G, s) e che, quindi, venga eseguita sugli archi di G una sequenza di passi di rilassamento che comprende la chiamata RELAX(u, v, w). Se ad un qualunque istante prima della chiamata vale $d[u] = \delta(s, u)$, allora in un qualunque istante dopo la chiamata varrà $d[v] = \delta(s, v)$.

Dimostrazione. Per il lemma 25.5, se $d[u] = \delta(s, u)$ in qualche istante prima del rilassamento dell'arco (u, v), allora questa uguaglianza continua a valere nel seguito. In particolare, dopo aver rilassato l'arco (u, v), abbiamo

$$\begin{aligned} d[v] &\le d[u] + w(u, v) \qquad \text{(per il lemma 25.4)} \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) \qquad \text{(per il corollario 25.2)} . \end{aligned}$$

Per il lemma 25.5, $\delta(s, v)$ è un limite inferiore per $d[v]$, da cui possiamo concludere che $d[v] = \delta(s, v)$ e questa uguaglianza viene mantenuta nel seguito. ■

Alberi di cammini minimi

Finora abbiamo mostrato che il rilassamento fa decrescere le stime di cammino minimo monotonamente verso gli effettivi pesi di cammino minimo. Vogliamo anche mostrare che, una volta che la sequenza di rilassamenti ha calcolato i reali pesi di cammino minimo, il sottografo dei predecessori G_π indotto dai valori π risultanti è un albero di cammini minimi per G. Iniziamo con il lemma seguente che mostra che il sottografo dei predecessori forma sempre un albero la cui radice è la sorgente.

Lemma 25.8

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w : E \to \mathbf{R}$ e con vertice sorgente $s \in V$ e si assuma che G non contenga cicli di peso negativo raggiungibili da s. Allora, dopo che il grafo è stato inizializzato con INITIALIZE-SINGLE-SOURCE(G, s), il sottografo dei predecessori G_π forma un albero radicato avente s come radice e qualunque sequenza di passi di rilassamento sugli archi di G mantiene questa proprietà invariante.

Dimostrazione. All'inizio, l'unico vertice in G_π è il vertice sorgente, quindi il lemma è banalmente vero. Si consideri ora il sottografo dei predecessori G_π prodotto da una sequenza di passi di rilassamento: come prima cosa dimostriamo che G_π è aciclico. Si supponga per assurdo che un qualche passo di rilassamento crei un ciclo nel grafo G_π, e che il ciclo sia

$c = \langle v_0, v_1, \ldots, v_k \rangle$, dove $v_k = v_0$. Allora $\pi[v_i] = v_{i-1}$ per $i = 1, 2, \ldots, k$ e, senza perdita di generalità, possiamo assumere che sia stato il rilassamento dell'arco $\langle v_{k-1}, v_k \rangle$ a creare il ciclo in G_π.

Non è difficile convincersi che tutti i vertici del ciclo c sono raggiungibili dalla sorgente s: infatti, tutti i vertici in c hanno un predecessore diverso da NIL e quindi ad ogni vertice in c è stata assegnata una stima di cammino minimo finita allorché gli è stato assegnato un valore π diverso da NIL. Per il lemma 25.5, ogni vertice nel ciclo c ha un peso di cammino minimo finito e questo implica che esso è raggiungibile da s.

Consideriamo ora le stime di cammino minimo su c prima della chiamata di RELAX(v_{k-1}, v_k, w) e mostriamo che c è un ciclo di peso negativo, contraddicendo quindi l'ipotesi che G non contenga cicli di peso negativo raggiungibili dalla sorgente. Subito prima della chiamata, abbiamo che $\pi[v_i] = v_{i-1}$, per $i = 1, 2, \ldots, k-1$. Quindi, per $i = 1, 2, \ldots, k-1$, l'ultimo aggiornamento di $d[v_i]$ è stato effettuato con l'assegnamento $d[v_i] \rightarrow d[v_{i-1}] + w(v_i, v_{i-1})$; inoltre, se $d[v_{i-1}]$ è cambiato da allora, può solo essere diminuito. Quindi, subito prima della chiamata di RELAX(v_{k-1}, v_k, w) si ha che

$$d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i), \text{ per tutti gli } i = 1, 2, \ldots, k-1. \tag{25.1}$$

Poiché $\pi[v_k]$ viene modificato dalla chiamata, subito prima vale anche la disuguaglianza stretta

$$d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k) .$$

Sommando questa disuguaglianza con le $k-1$ disuguaglianze (25.1), si ottiene la somma delle stime di cammino minimo lungo il ciclo c:

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &> \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) . \end{aligned}$$

Ma poiché ogni vertice nel ciclo c compare esattamente una volta in ogni sommatoria, abbiamo che

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$$

e questo implica

$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Quindi, la somma dei pesi lungo il ciclo c è negativa e questo fornisce la contraddizione cercata. ■

Abbiamo quindi dimostrato che G_π è un grafo orientato e aciclico: per mostrare che esso forma un albero con radice s, è sufficiente dimostrare che per ogni vertice $v \in V_\pi$ vi è un unico cammino da s a v in G_π (si veda l'Esercizio 5.5-3).

Come prima cosa, occorre dimostrare che esiste un cammino da s ad ogni vertice in V_π, dove i vertici in V_π sono quelli con valore π diverso da NIL, più s. Questa dimostrazione si può fare per induzione: i dettagli sono lasciati come esercizio al lettore (Esercizio 25.1-6).

Per completare la dimostrazione del lemma, dobbiamo mostrare che, per ogni vertice $v \in V_\pi$, vi è al massimo un cammino da s a v nel grafo G_π. Si supponga che ciò non sia vero, cioè che

Figura 25.4 *Un cammino in G_π dalla sorgente s al vertice v è unico. Se vi fossero due cammini p_1 ($s \leadsto u \leadsto x \rightarrow z \leadsto v$) e p_2 ($s \leadsto u \leadsto y \rightarrow z \leadsto v$), con $x \neq y$, allora si avrebbe $\pi[z] = x$ e $\pi[z] = y$, il che è una contraddizione.*

vi siano due cammini semplici p_1 e p_2 da s ad un qualche vertice v, precisamente p_1, che può essere decomposto in $s \leadsto u \leadsto x \rightarrow z \leadsto v$ e p_2, che può essere decomposto in $s \leadsto u \leadsto y \rightarrow z \leadsto v$, con $x \neq y$ (si veda la figura 25.4). Ma allora $\pi[z] = x$ e $\pi[z] = y$, e ciò implica $x = y$, dando luogo ad una contraddizione. Quindi, si può concludere che esiste un unico cammino semplice in G_π da s a v e quindi che G_π forma un albero con radice s. ■

Possiamo ora mostrare che se, dopo aver eseguito una sequenza di passi di rilassamento, ad ogni vertice è stato assegnato il proprio peso di cammino minimo reale, allora il sottografo dei predecessori G_π è un albero di cammini minimi.

Lemma 25.9

Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w: E \rightarrow \mathbf{R}$ e vertice sorgente $s \in V$, e si assuma che G non contenga cicli di peso negativo raggiungibili da s. Si invochi la procedura INITIALIZE-SINGLE-SOURCE(G, s) e quindi si esegua una qualunque sequenza di passi di rilassamento sugli archi di G che produca $d[v] = \delta(s, v)$ per ogni $v \in V$. Allora, il sottografo dei predecessori G_π è un albero di cammini minimi con radice s.

Dimostrazione. Dobbiamo dimostrare che le tre proprietà degli alberi di cammini minimi valgono per G_π. Per mostrare la prima proprietà, occorre mostrare che V_π è l'insieme di tutti i vertici raggiungibili da s. Per definizione, un peso di cammino minimo $\delta(s, v)$ è finito se e solo se v è raggiungibile da s, quindi i vertici che sono raggiungibili da s sono esattamente quelli con valore d finito. Ma ad un vertice $v \in V - \{s\}$ viene assegnato un valore finito per $d[v]$ se e solo se $\pi[v] \neq$ NIL, quindi i vertici in V_π sono esattamente quelli raggiungibili da s.

La seconda proprietà segue direttamente dal lemma 25.8.

Rimane quindi da dimostrare l'ultima proprietà degli alberi di cammini minimi: per tutti i $v \in V$, l'unico cammino semplice $s \leadsto v$ in G_π è un cammino minimo da s a v in G. Sia $p = \langle v_0, v_1, \ldots, v_k \rangle$, con $v_0 = s$ e $v_k = v$. Per ogni $i = 1, 2, \ldots, k$, abbiamo sia $d[v_i] = \delta(s, v_i)$, sia $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$, da cui possiamo concludere che $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$. Sommando i pesi lungo il cammino p otteniamo

$$\begin{aligned} w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) \\ &= \delta(s, v_k) . \end{aligned}$$

La terza linea segue dalla somma telescopica nella seconda linea, mentre la quarta linea segue da $\delta(s, v_0) = \delta(s, s) = 0$. Quindi, abbiamo mostrato che $w(p) \le \delta(s, v_k)$. Poiché $\delta(s, v_k)$ è un limite inferiore per il peso di un qualunque cammino da s a v_k, ne segue che $w(p) = \delta(s, v_k)$ e quindi p è un cammino minimo da s a $v = v_k$. ■

Esercizi

25.1-1 Si diano due alberi di cammini minimi per il grafo orientato di figura 25.2, diversi da quelli mostrati.

25.1-2 Si dia un esempio di un grafo orientato e pesato $G = (V, E)$ con funzione peso $w : E \to \mathbf{R}$ e sorgente s tale che G soddisfi la seguente proprietà: per ogni arco $(u, v) \in E$ vi è un albero di cammini minimi con radice s che contiene (u, v), ed un altro albero di cammini minimi con radice s che non contiene (u, v).

25.1-3 Si migliori la dimostrazione del lemma 25.3 per gestire i casi in cui i pesi di cammino minimo siano ∞ o $-\infty$.

25.1-4 Sia $G = (V, E)$ un grafo orientato e pesato con vertice sorgente s e si assuma che G sia inizializzato con INIZIALIZE-SINGLE-SOURCE(G, s). Si dimostri che se una sequenza di passi di rilassamento assegna a $\pi[s]$ un valore diverso da NIL, allora G contiene un ciclo di peso negativo.

25.1-5 Sia $G = (V, E)$ un grafo orientato e pesato senza archi di peso negativo. Sia $s \in V$ il vertice radice e si definisca $\pi[v]$ in modo usuale: $\pi[v]$ è il predecessore di v su di un qualche cammino minimo da s a v, se $v \in V - \{s\}$, NIL altrimenti. Si dia un esempio di un tale grafo G, ed un assegnamento dei valori π che produca un ciclo in G_π. (Per il lemma 25.8, un tale assegnamento non può essere prodotto da una sequenza di passi di rilassamento).

25.1-6 Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w : E \to \mathbf{R}$ e senza cicli di peso negativo. Sia $s \in V$ il vertice sorgente e si supponga che G sia inizializzato con INIZIALIZE-SINGLE-SOURCE(G, s). Si dimostri che per ogni vertice $v \in V_\pi$ vi è un cammino da s a v in G_π e che questa proprietà viene mantenuta invariante da qualunque sequenza di passi di rilassamento.

25.1-7 Sia $G = (V, E)$ un grafo orientato e pesato senza cicli di peso negativo. Sia $s \in V$ il vertice sorgente e si supponga che G sia inizializzato con INIZIALIZE-SINGLE-SOURCE(G, s). Si dimostri che vi è una sequenza di $|V| - 1$ passi di rilassamento che producono $d[v] = \delta(s, v)$ per ogni $v \in V$.

25.1-8 Sia G un arbitrario grafo orientato e pesato con un ciclo di peso negativo raggiungibile dal vertice sorgente s. Si mostri che è sempre possibile costruire una sequenza infinita di rilassamenti degli archi di G tale che ogni rilassamento modifica una stima di cammino minimo.

25.2 Algoritmo di Dijkstra

L'algoritmo di Dijkstra risolve il problema di cammini minimi con sorgente singola su un grafo orientato e pesato $G = (V, E)$ nel caso in cui tutti i pesi degli archi siano non negativi. In questo paragrafo assumiamo quindi che $w(u, v) \ge 0$ per ogni arco $(u, v) \in E$.

L'algoritmo di Dijkstra mantiene un insieme S che contiene i vertici il cui peso di cammino minimo dalla sorgente s è già stato determinato, cioè, per tutti i vertici $v \in S$, vale $d[v] = \delta(s, v)$. L'algoritmo seleziona ripetutamente il vertice $u \in V - S$ con la minima stima di cammino minimo, inserisce u in S, e rilassa tutti gli archi uscenti da u. Nella realizzazione che segue, si mantiene una coda a priorità Q che contiene tutti i vertici in $V - S$, usando come chiave i rispettivi valori d; la realizzazione assume che il grafo G sia rappresentato con liste di adiacenza.

```
DIJKSTRA(G, w, s)
1  INIZIALIZE-SINGLE-SOURCE(G, s)
2  S ← ∅
3  Q ← V[G]
4  while Q ≠ ∅
5      do u ← EXTRACT-MIN(Q)
6          S ← S ∪ {u}
7          for ogni vertice v ∈ Adj[u]
8              do RELAX(u, v, w)
```

L'algoritmo di Dijkstra rilassa gli archi come mostrato nella figura 25.5. La linea 1 esegue la normale inizializzazione dei valori d e π, la linea 2 inizializza l'insieme S con l'insieme vuoto. La linea 3 inizializza la coda a priorità Q con tutti i vertici in $V - S = V - \varnothing = V$. Ogni volta che si esegue il ciclo **while** delle linee 4-8, un vertice u viene estratto da $Q = V - S$ e viene inserito nell'insieme S (la prima volta che viene eseguito il ciclo, si ha $u = s$). Il vertice u ha quindi la minima stima di cammino minimo tra tutti i vertici in $V - S$. Quindi, le linee 7-8 rilassano ogni arco (u, v) che esce da u, aggiornando la stima $d[v]$ ed il predecessore $\pi[v]$ se il cammino minimo per v può essere migliorato passando per u. Si osservi che nessun vertice viene inserito in Q dopo la linea 3 e che ogni vertice viene estratto da Q ed inserito in S esattamente una volta; quindi, il ciclo **while** delle linee 4-8 viene ripetuto esattamente $|V|$ volte.

Poiché l'algoritmo di Dijkstra sceglie sempre il vertice in $V - S$ "più leggero" o "più vicino" da inserire in S, diremo che esso usa una strategia greedy ("golosa"). Le strategie greedy sono presentate in dettaglio nel Capitolo 17, ma non è necessario aver letto quel capitolo per comprendere l'algoritmo di Dijkstra. In generale, le strategie greedy non producono sempre risultati ottimali, ma come mostrano il seguente teorema ed il suo corollario, l'algoritmo di Dijkstra calcola effettivamente i cammini minimi; il punto chiave consiste nel mostrare che quando un vertice u viene inserito nell'insieme S si ha $d[u] = \delta(s, u)$.

Teorema 25.10 (Correttezza dell'algoritmo di Dijkstra)

Se si esegue l'algoritmo di Dijkstra su un grafo orientato e pesato $G = (V, E)$ con funzione peso non negativa w e sorgente s, allora al termine vale $d[u] = \delta(s, u)$ per ogni vertice $u \in V$.

Figura 25.5 *Un'esecuzione dell'algoritmo di Dijkstra. La sorgente è il vertice più a sinistra. Le stime di cammino minimo sono indicate all'interno dei vertici e gli archi grigi indicano i valori del campo predecessore: se l'arco (u, v) è grigio, allora $\pi[v] = u$. I vertici neri sono nell'insieme S, mentre i vertici bianchi, sono nella coda a priorità $Q = V - S$.* **(a)** *La situazione subito prima della prima iterazione del ciclo* **while** *delle linee 4-8. Il vertice in grigio ha il valore minimo di d ed è scelto come vertice u nella linea 5.* **(b)-(f)** *la situazione dopo ogni iterazione successiva del ciclo* **while**. *Il vertice grigio in ogni fase è scelto come vertice u nella linea 5 della prossima iterazione. I valori d e π mostrati nella parte* **(f)** *sono valori finali.*

Dimostrazione. Mostreremo che, per ogni vertice $u \in V$, si ha $d[u] = \delta(s, u)$ al momento in cui u viene inserito nell'insieme S, e che questa uguaglianza viene mantenuta nel seguito.

Si supponga, per assurdo, che u sia il primo vertice per il quale $d[u] \neq \delta(s, u)$ quando esso viene inserito nell'insieme S. Focalizzeremo l'attenzione sulla situazione all'inizio dell'iterazione del ciclo **while** nel quale u viene inserito in S, ed esaminando un cammino minimo da s ad u deriveremo la contraddizione che in quel momento $d[u] = \delta(s, u)$. Naturalmente, deve valere $u \neq s$, perché s è il primo vertice che viene inserito in s, ed in quel momento vale $d[s] = \delta(s, s) = 0$: poiché $u \neq s$, abbiamo anche che $s \neq \varnothing$ subito prima che u venga inserito in S. Inoltre, deve esistere un cammino da s a u, altrimenti $d[u] = \delta(s, u) = \infty$ per il corollario 25.6, il che violerebbe l'assunzione $d[u] \neq \delta(s, u)$; siccome vi è almeno un cammino da s a u, allora deve esistere un cammino minimo p da s a u. Il cammino p collega un vertice in S, cioè s, ad un vertice in $V - S$, cioè u; sia y il primo vertice lungo p tale che $y \in V - S$, e sia $x \in S$ il predecessore di y. Allora, il cammino p può essere decomposto come $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, come mostrato in figura 25.6.

Mostriamo ora che $d[y] = d(s, y)$ quando u viene inserito in S. Infatti, poiché $x \in s$ ed u è scelto come il primo vertice per cui $d[u] \neq \delta(s, u)$ quando esso viene inserito in S, si deve avere che $d[x] = \delta(s, x)$ quando x è stato inserito in S. Ma l'arco (x, y) è stato rilassato proprio in quel momento e quindi, dal lemma 25.7 segue che $d[y] = \delta(s, y)$ quando u viene inserito in S.

Possiamo ora ottenere la contraddizione che dimostra il teorema. Infatti, poiché y compare prima di u in un cammino minimo da s ad u e, poiché tutti i pesi degli archi sono non negativi (in particolare quelli sul cammino p_2), abbiamo che $\delta(s, y) \leq \delta(s, u)$ e quindi

Figura 25.6 *La dimostrazione del Teorema 25.10. L'insieme S è non vuoto subito prima che il vertice u venga inserito in esso. Un cammino minimo p dalla sorgente s al vertice u può essere decomposto in $s \overset{p_1}{\leadsto} x \to y \overset{p_2}{\leadsto} u$, dove y è il primo vertice nel cammino che non è in S e $x \in S$ precede immediatamente y. I vertici x e y sono distinti, ma si può avere $s = x$ o $y = u$. Il cammino p_2 può eventualmente rientrare nell'insieme S.*

$$\begin{aligned} d[y] &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq d[u] \quad \text{(per il lemma 25.5)} \end{aligned} \tag{25.2}$$

Ma poiché entrambi i vertici u e y erano in $V - S$ quando u era stato scelto nella linea 5, si ha necessariamente che $d[u] \leq d[y]$, quindi le due disuguaglianze in (25.2) sono in effetti delle uguaglianze:

$$d[y] = \delta(s, y) = \delta(s, u) = d[u].$$

Di conseguenza $d[u] = \delta(s, u)$, il che contraddice la scelta iniziale di u. Possiamo quindi concludere che nel momento in cui un qualunque vertice $u \in V$ viene inserito in S, deve valere $d[u] = \delta(s, u)$ e, per il lemma 25.5, questa uguaglianza deve continuare a valere nel seguito. ■

Corollario 25.11

Se si esegue l'algoritmo di Dijkstra su un grafo orientato e pesato $G = (V, E)$ con funzione peso non negativa w e sorgente s, allora al termine il sottografo dei predecessori G_π è un albero di cammini minimi con radice s.

Dimostrazione. Immediata per il Teorema 25.10 e per il lemma 25.9. ■

Analisi

Quanto è veloce l'algoritmo di Dijkstra? Si consideri dapprima il caso in cui si mantenga la coda con priorità $Q = V - S$ come un array lineare. Per questa realizzazione, ogni operazione EXTRACT-MIN richiede tempo $O(V)$ e, poiché vi sono $|V|$ operazioni di questo tipo, il tempo totale richiesto da EXTRACT-MIN è $O(V^2)$. Ogni vertice $v \in V$ viene inserito nell'insieme S esattamente una volta e quindi ogni arco nella lista di adiacenza $Adj[v]$ viene esaminato nel ciclo **for** delle linee 7-8 esattamente una volta nel corso dell'algoritmo. Poiché il numero totale di archi in tutte le liste di adiacenza è $|E|$, vi sono in totale $|E|$ iterazioni di questo ciclo **for**, ognuna delle quali richiede tempo $O(1)$. Quindi, il tempo totale di esecuzione dell'algoritmo è $O(V^2 + E) = O(V^2)$.

Se però il grafo è sparso, può essere conveniente realizzare la coda a priorità Q con uno heap binario: l'algoritmo risultante è chiamato talvolta ***algoritmo di Dijkstra modificato***. Ogni operazione EXTRACT-MIN richiede allora tempo $O(\lg V)$; come prima, vi sono $|V|$ di queste operazioni. Il tempo necessario per costruire lo heap binario è $O(V)$. L'assegnamento $d[v] \leftarrow d[u] + w(u, v)$ in RELAX viene effettuato dalla chiamata DECREASE-KEY$(Q, v, d[u] + w(u, v))$, che richiede tempo $O(\lg V)$ (si veda l'Esercizio 7.5-4), e vi sono al massimo $|E|$ di queste operazioni. Quindi il tempo totale di esecuzione è $O((V+E)\lg V)$ che diventa $O(E \lg V)$ se tutti i vertici sono raggiungibili dalla sorgente.

In effetti, si può ottenere un tempo di esecuzione $O(V \lg V + E)$ realizzando la coda a priorità Q con uno heap di Fibonacci (si veda il Capitolo 21). Il costo ammortizzato di ognuna delle $|V|$ operazioni EXTRACT-MIN è $O(\lg V)$ ed ognuna delle $|E|$ chiamate a DECREASE-KEY richiede solo un tempo ammortizzato $O(1)$. Storicamente, lo sviluppo degli heap di Fibonacci è stato motivato dall'osservazione che nell'algoritmo di Dijkstra modificato vi sono potenzialmente molte più chiamate a DECREASE-KEY che a EXTRACT-MIN e che, quindi, ogni metodo capace di ridurre il tempo ammortizzato di ogni operazione DECREASE-KEY a $o(\lg V)$, senza aumentare il tempo ammortizzato di EXTRACT-MIN, avrebbe portato ad un tempo di calcolo asintoticamente più efficiente.

L'algoritmo di Dijkstra ha delle analogie sia con la visita in ampiezza (si veda il paragrafo 23.2) che con l'algoritmo di Prim per calcolare gli alberi di copertura minimi (paragrafo 24.2). Esso è simile alla visita in ampiezza, nel senso che l'insieme S corrisponde all'insieme dei vertici neri in una visita in ampiezza: proprio come i vertici in S hanno al termine il loro peso di cammino minimo, così i vertici neri in una visita in ampiezza hanno al termine le loro distanze BFS corrette. L'algoritmo di Dijkstra è simile all'algoritmo di Prim perché entrambi gli algoritmi usano una coda a priorità per trovare il vertice "più leggero" che non appartenga ad un dato insieme (l'insieme S nell'algoritmo di Dijkstra, e l'albero che viene costruito nell'algoritmo di Prim), inseriscono questo vertice nell'insieme e modificano adeguatamente i pesi dei rimanenti vertici che non appartengono all'insieme.

Esercizi

25.2-1 Si esegua l'algoritmo di Dijkstra sul grafo orientato di figura 25.2 usando prima il vertice s e poi il vertice y come sorgente. Nello stile della figura 25.5, si mostrino i valori d e π ed i vertici nell'insieme S dopo ogni iterazione del ciclo **while**.

25.2-2 Si dia un semplice esempio di un grafo orientato con archi di peso negativo per il quale l'algoritmo di Dijkstra produce un risultato scorretto. Perché la dimostrazione del Teorema 25.10 non può essere applicata se sono ammessi archi con peso negativo?

25.2-3 Si supponga di cambiare la linea 4 dell'algoritmo di Dijkstra con la seguente:

4 **while** $|Q| > 1$

Questo cambio fa eseguire il ciclo $|V| - 1$ volte invece di $|V|$. L'algoritmo proposto è corretto?

25.2-4 Sia dato un grafo orientato $G = (V, E)$ nel quale ad ogni arco $(u, v) \in E$ è associato un valore reale $r[u, v] \in [0, 1]$, che rappresenta l'affidabilità di un canale di comunicazione dal vertice u al vertice v. Si interpreti $r(u, v)$ come la probabilità che il canale da u a v trasmetta correttamente un messaggio e si assuma che queste probabilità siano indipendenti. Si dia un algoritmo efficiente per trovare il cammino più affidabile tra due vertici dati.

25.2-5 Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w : E \rightarrow \{0, 1, \ldots, W - 1\}$, con W intero non negativo. Si modifichi l'algoritmo di Dijkstra per calcolare i cammini minimi da un dato vertice sorgente s in tempo $O(WV + E)$.

25.2-6 Si modifichi l'algoritmo proposto per l'Esercizio 25.2-5 in modo che abbia tempo di esecuzione $O((V + E) \lg W)$. (*Suggerimento*: quante stime di cammino minimo distinte vi possono essere in $V - S$ ad un qualunque istante?)

25.3 Algoritmo di Bellman-Ford

L'algoritmo di Bellman-Ford risolve il problema di cammini minimi con sorgente singola nel caso più generale in cui i pesi degli archi possono essere negativi. Dato un grafo orientato e pesato $G = (V, E)$ con sorgente s e funzione peso $w : E \rightarrow \mathbf{R}$, l'algoritmo di Bellman-Ford restituisce un valore booleano che indica se esiste oppure no un ciclo di peso negativo raggiungibile dalla sorgente. In caso affermativo, l'algoritmo indica che non esiste alcuna soluzione; se invece un tale ciclo non esiste, allora l'algoritmo produce i cammini minimi ed i loro pesi.

Come l'algoritmo di Dijkstra, anche l'algoritmo di Bellman-Ford usa la tecnica del rilassamento, diminuendo progressivamente una stima $d[v]$ del peso di un cammino minimo dalla sorgente s ad ogni vertice $v \in V$ fino a raggiungere il reale peso di cammino minimo $\delta(s, v)$. L'algoritmo restituisce TRUE se e solo se il grafo non contiene un ciclo di peso negativo raggiungibile dalla sorgente.

```
BELLMAN-FORD(G, w, s)
1  INIZIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to |V[G]| − 1
3      do for ogni vertice (u, v) ∈ E[G]
4          do RELAX(u, v, w)
5  for ogni arco (u, v) ∈ E[G]
6      do if d[v] > d[u] + w(u, v)
7          then return FALSE
8  return TRUE
```

La figura 25.7 mostra l'esecuzione dell'algoritmo di Bellman-Ford su un grafo con 5 vertici. Dopo aver effettuato la solita inizializzazione, l'algoritmo fa $|V| - 1$ passate sugli archi del grafo; ogni passata è una iterazione del ciclo **for** delle linee 2-4 e consiste nel rilassare ogni

Figura 25.7 *Un'esecuzione dell'algoritmo di Bellman-Ford. La sorgente è il vertice z. I valori d sono mostrati all'interno dei vertici e gli archi in grigio indicano i valori π. In questo specifico esempio, ogni passata rilassa gli archi in ordine lessicografico: (u, v), (u, x), (u, y), (v, u), (x, v), (x, y), (y, v), (y, z), (z, u), (z, x).* (a) *La situazione subito prima della prima passata sugli archi.* (b)-(e) *La situazione dopo ogni passata successiva sugli archi. I valori d e π nella parte* (e) *sono i valori finali. L'algoritmo di Bellman-Ford, in questo esempio, restituisce* TRUE.

arco del grafo una volta. Le figure 25.7(b)-(e) mostrano lo stato dell'algoritmo dopo ognuna delle quattro passate sugli archi. Dopo le $|V| - 1$ passate, le linee 5-8 controllano l'esistenza di un ciclo di peso negativo e restituiscono il valore booleano appropriato (vedremo tra poco come funziona questo controllo).

L'algoritmo di Bellman-Ford richiede tempo $O(VE)$, poiché l'inizializzazione in linea 1 richiede tempo $\Theta(V)$, ognuna delle $|V| - 1$ passate sugli archi nelle linee 2-4 richiede tempo $O(E)$, ed il ciclo **for** delle linee 5-7 richiede tempo $O(E)$.

Per dimostrare la correttezza dell'algoritmo di Bellman-Ford, cominciamo col mostrare che se non vi sono cicli di peso negativo, l'algoritmo calcola i pesi corretti di cammino minimo per tutti i vertici raggiungibili dalla sorgente. La dimostrazione di questo lemma contiene l'intuizione su cui si basa l'algoritmo.

Lemma 25.12

Sia $G = (V, E)$ un grafo orientato e pesato con sorgente s e funzione peso $w: E \to \mathbf{R}$ e si assuma che G non contenga cicli di peso negativo raggiungibili da s. Allora, al termine di BELLMAN-FORD, vale $d[v] = \delta(s, v)$ per tutti i vertici v raggiungibili da s.

Dimostrazione. Sia v un vertice raggiungibile da s, e sia $p = \langle v_0, v_1, \ldots, v_k \rangle$ un cammino minimo da s a v, dove $v_0 = s$ e $v_k = v$; il cammino p è semplice, quindi $k \le |V| - 1$. Vogliamo dimostrare per induzione che, per $i = 0, 1, \ldots, k$, si ha $d[v_i] = \delta(s, v_i)$ dopo l'i-esima passata sugli archi di G, e che questa uguaglianza viene mantenuta nel seguito; poiché vi sono $|V| - 1$ passate, questo è sufficiente per dimostrare il lemma.

Per la base dell'induzione, abbiamo che dopo l'inizializzazione $d[v_0] = \delta(s, v_0) = 0$; per il lemma 25.5 questa uguaglianza viene mantenuta nel seguito.

Per il passo induttivo, si assuma che $d[v_{i-1}] = \delta(s, v_{i-1})$ dopo l'$(i-1)$-esima passata; poiché l'arco (v_{i-1}, v_i) viene rilassato nell'i-esima passata, dal lemma 25.7 si conclude che $d[v_i] = \delta(s, v_i)$ dopo l'i-esima passata ed in ogni istante successivo, il che completa la dimostrazione. ■

Corollario 25.13

Sia $G = (V, E)$ un grafo orientato e pesato con sorgente s e funzione peso $w: E \to \mathbf{R}$. Allora per ogni vertice $v \in V$ vi è un cammino da s a v se e solo se BELLMAN-FORD, quando viene eseguito su G, termina con $d[v] < \infty$.

Dimostrazione. La dimostrazione è simile a quella del lemma 25.12 ed è lasciata al lettore per esercizio (Esercizio 25.3-2). ■

Teorema 25.14 (Correttezza dell'algoritmo di Bellman-Ford)

Si esegua BELLMAN-FORD su di un grafo orientato e pesato $G = (V, E)$ con sorgente s e funzione peso $w: E \to \mathbf{R}$. Se G non contiene cicli di peso negativo raggiungibili da s, allora l'algoritmo restituisce TRUE, per tutti i vertici $v \in V$ si ha $d[v] = \delta(s, v)$ e il sottografo dei predecessori G_π è un albero di cammini minimi con s come radice. Se invece G contiene un ciclo di peso negativo raggiungibile da s, allora l'algoritmo restituisce FALSE.

Dimostrazione. Si supponga che il grafo G non contenga cicli di peso negativo raggiungibili dalla sorgente s. Come prima cosa mostriamo che, al termine, per tutti i vertici $v \in V$ vale $d[v] = \delta(s, v)$: ciò è garantito dal lemma 25.12 se il vertice v è raggiungibile da s, e dal corollario 25.6 se v non è raggiungibile da s. Il lemma 25.9 ed il fatto che $d[v] = \delta(s, v)$ per ogni $v \in V$ garantiscono che G_π è un albero di cammini minimi. Resta da dimostrare che BELLMAN-FORD restituisce TRUE: quando l'algoritmo termina, per ogni $(u, v) \in E$ abbiamo che

$$\begin{aligned} d[v] &= \delta(s, v) \\ &\le \delta(s, u) + w(u, v) \qquad \text{(per il lemma 25.3)} \\ &= d[u] + w(u, v) , \end{aligned}$$

e quindi nessuno dei controlli nella linea 6 può fare in modo che BELLMAN-FORD restituisca FALSE.

Si supponga ora che il grafo G contenga un ciclo di peso negativo $c = \langle v_0, v_1, \ldots, v_k \rangle$, con $v_0 = v_k$, e che c sia raggiungibile dalla sorgente s. Allora

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 , \tag{25.3}$$

Si assuma, per assurdo, che l'algoritmo di Bellman-Ford restituisca TRUE.

Allora $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$, per $i = 1, 2, \ldots, k$. Sommando queste disuguaglianze lungo il ciclo c si ottiene

$$\sum_{i=1}^{k} d[v_i] \le \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Come nella dimostrazione del lemma 25.8, ogni vertice di c compare esattamente una volta in ognuna delle due prime sommatorie, quindi

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] .$$

Inoltre, per il corollario 25.13, $d[v_i]$ è finito per $i = 1, 2, \ldots, k$, quindi

$$0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i) ,$$

che contraddice la disuguaglianza (25.3). Si può dunque concludere che l'algoritmo di Bellman-Ford restituisce TRUE se il grafo G non contiene cicli di peso negativo raggiungibili dalla sorgente, FALSE altrimenti. ■

Esercizi

25.3-1 Si esegua l'algoritmo di Bellman-Ford sul grafo orientato della figura 25.7, usando il vertice y come sorgente. Si rilassino gli archi in ordine lessicografico ad ogni passata e si mostrino i valori d e π dopo ogni passata. Quindi, si cambi il peso dell'arco (y, v) a 4 e si esegua di nuovo l'algoritmo con z come sorgente.

25.3-2 Si dimostri il corollario 25.13.

25.3-3 Dato un grafo orientato e pesato $G = (V, E)$ senza cicli di peso negativo, sia m il massimo (per tutte le coppie di vertici $u, v \in V$) tra i minimi numeri di archi su un cammino minimo da u a v (qui si intende cammino minimo rispetto al peso e non rispetto al numero degli archi). Si suggerisca una semplice modifica all'algoritmo di Bellman-Ford che gli consenta di terminare in $m + 1$ passate.

25.3-4 Si modifichi l'algoritmo di Bellman-Ford in modo tale che $d[v]$ venga posto a $-\infty$ per tutti quei vertici v per i quali vi sia un ciclo di peso negativo su di un cammino dalla sorgente a v.

25.3-5 Sia $G = (V, E)$ un grafo orientato e pesato con funzione peso $w : E \to \mathbf{R}$. Si dia un algoritmo in tempo $O(VE)$ per trovare, per ogni vertice $v \in V$, il valore $\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}$.

* *25.3-6* Si supponga che un grafo orientato e pesato $G = (V, E)$ abbia almeno un ciclo di peso negativo. Si dia un algoritmo efficiente per elencare i vertici di uno di questi cicli e si dimostri la correttezza dell'algoritmo proposto.

25.4 Cammini minimi con sorgente singola in grafi orientati aciclici

Rilassando gli archi di un grafo orientato aciclico (dag) pesato $G = (V, E)$ secondo un ordinamento topologico dei suoi vertici, si possono calcolare i cammini minimi da una singola sorgente in tempo $\Theta(V + E)$. I cammini minimi sono sempre ben definiti in un dag perché, anche se esistono archi di peso negativo, non possono esistere cicli di peso negativo.

L'algoritmo inizia ordinando topologicamente il dag (si veda il paragrafo 23.4) per imporre un ordinamento lineare sui vertici: se vi è un cammino dal vertice u al vertice v allora u precede v nell'ordinamento topologico. Si esegue quindi una sola passata sui vertici del grafo seguendo l'ordinamento topologico: quando un vertice viene elaborato, tutti gli archi uscenti da esso vengono rilassati.

```
DAG-SHORTEST-PATHS(G, w, s)
1  si pongano i vertici di G in ordine topologico
2  INIZIALIZE-SINGLE-SOURCE(G, s)
3  for ogni vertice u preso secondo l'ordinamento topologico
4      do for ogni vertice v ∈ Adj[u]
5          do RELAX(u, v, w)
```

Un esempio dell'esecuzione di questo algoritmo è mostrato nella figura 25.8.

Il tempo di esecuzione di questo algoritmo è determinata dalla linea 1 e dal ciclo **for** delle linee da 3 a 5. Come mostrato nel paragrafo 23.4, l'ordinamento topologico può essere eseguito in tempo $\Theta(V + E)$. Nel ciclo **for** delle linee 3-5, come nell'algoritmo di Dijkstra, vi è una iterazione per ogni vertice: in ogni iterazione gli archi uscenti dal vertice vengono esaminati esattamente una volta. Tuttavia, a differenza dell'algoritmo di Dijkstra, non esiste coda a priorità, quindi si usa solo un tempo $O(1)$ per ogni arco. Il tempo di esecuzione totale dell'algoritmo è perciò $\Theta(V + E)$, che è lineare nella dimensione di una rappresentazione del grafo con liste di adiacenza.

Il seguente teorema mostra che la procedura DAG-SHORTEST-PATHS calcola correttamente i cammini minimi.

Teorema 25.15

Se un grafo orientato e pesato $G = (V, E)$ ha vertice sorgente s e non ha cicli, allora, al termine della procedura DAG-SHORTEST-PATHS, vale $d[v] = \delta(s, v)$ per ogni vertice $v \in V$ ed il sottografo dei predecessori G_π è un albero di cammini minimi.

Dimostrazione. Innanzitutto, dimostriamo che $d[v] = \delta(s, v)$, per tutti i vertici $v \in V$, quando l'algoritmo termina. Se v non è raggiungibile da s, allora $d[v] = \delta(s, v) = \infty$ per il corollario 25.6. Se invece v è raggiungibile da s, allora vi è un cammino minimo $p = \langle v_0, v_1, \ldots, v_k \rangle$, dove $v_0 = s$ e $v_k = v$. Poiché i vertici vengono elaborati in ordine topologico, gli archi in p vengono rilassati nell'ordine $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$: una semplice induzione usando il lemma 25.7 (come nella dimostrazione del lemma 25.12) mostra che al termine vale $d[v_i] = \delta(s, v_i)$, per $i = 0, 1, \ldots, k$. Infine, per il lemma 25.9, G_π è un albero di cammini minimi. ■

Figura 25.8 *L'esecuzione dell'algoritmo per i cammini minimi in un grafo orientato aciclico. I vertici sono orientati topologicamente da sinistra a destra. Il vertice sorgente è s. I valori d sono mostrati all'interno dei vertici e gli archi grigi indicano i valori π.* (a) *La situazione precedente alla prima iterazione del ciclo* for *delle linee* 3-5. (b)-(g) *La situazione dopo ogni iterazione del ciclo* for *delle linee* 3-5. *In ogni iterazione, l'unico nuovo vertice colorato di nero (rispetto all'iterazione precedente) è quello usato come u nell'algoritmo. I valori mostrati nella parte* (g) *sono i valori finali.*

Questo algoritmo trova un'interessante applicazione nella determinazione dei cammini critici nell'analisi di ***diagrammi PERT***[2]. Gli archi rappresentano processi che devono essere eseguiti ed i pesi degli archi rappresentano i tempi richiesti per eseguire particolari processi. Se l'arco (u, v) entra nel vertice v e l'arco (v, x) ne esce, allora il processo (u, v) deve essere eseguito prima del processo (v, x). Un cammino in questo dag rappresenta una sequenza di processi che devono essere eseguiti in un particolare ordine. Un ***cammino critico*** è un cammino *massimo* nel dag, corrispondente al tempo massimo necessario per eseguire una sequenza ordinata di processi; il peso di un cammino critico è un limite inferiore al tempo totale necessario per eseguire tutti i processi. Si può trovare un cammino critico

- cambiando il segno ai pesi degli archi ed eseguendo DAG-SHORTEST-PATHS, oppure
- eseguendo DAG-SHORTEST-PATHS, sostituendo "∞" con "−∞" nella linea 2 di INIZIALIZE-SINGLE-SOURCE e ">" con "<" nella procedura RELAX.

[2] "PERT" è acronimo di "program evaluation and review technique".

Esercizi

25.4-1 Si esegua DAG-SHORTEST-PATHS sul grafo orientato della figura 25.8, usando il vertice r come sorgente.

25.4-2 Si cambi la linea 3 della procedura DAG-SHORTEST-PATHS in

3 **for** i primi $|V| - 1$ vertici presi in ordine topologico

Mostrare che la procedura rimane corretta.

25.4-3 La formulazione dei diagrammi PERT data precedentemente non è molto naturale. Sarebbe più naturale che i vertici rappresentassero i processi e gli archi i vincoli di sequenzializzazione, cioè l'arco (u, v) indicherebbe che il processo u deve essere eseguito prima del processo v. I pesi sarebbero allora associati ai vertici, non agli archi. Si modifichi la procedura DAG-SHORTEST-PATHS in modo che trovi il cammino massimo in un grafo orientato aciclico con vertici pesati in tempo lineare.

25.4-4 Si dia un algoritmo efficiente per contare il numero totale di cammini in un grafo orientato aciclico. Si analizzi l'algoritmo proposto e se ne commenti la praticabilità.

25.5 Vincoli di differenza e cammini minimi

Nel problema generale di programmazione lineare, si vuole ottimizzare una funzione lineare soggetta ad un insieme di disuguaglianze lineari. In questo paragrafo considereremo un caso speciale di programmazione lineare che può essere ridotto alla ricerca di cammini minimi da una singola sorgente. Il risultante problema di cammini minimi con sorgente singola può essere risolto con l'algoritmo di Bellman-Ford che risolve quindi anche il problema di programmazione lineare.

Programmazione lineare

Nel caso generale del ***problema di programmazione lineare*** viene data una matrice A di dimensione $m \times n$, un vettore b di lunghezza m ed un vettore c di lunghezza n. Si vuole trovare un vettore x di n elementi che massimizzi la ***funzione obiettivo*** $\sum_{i=1}^{n} c_i x_i$ soggetta agli m vincoli dati da $Ax \le b$.

Molti problemi possono essere espressi nell'ambito della programmazione lineare e per questo motivo molto è stato investito nello studio di algoritmi per la programmazione lineare. L'***algoritmo del simplesso***[3] risolve programmi lineari generali molto rapidamente, almeno nella pratica; tuttavia, su particolari input, il metodo del simplesso può richiedere tempo

[3] L'algoritmo del simplesso trova una soluzione ottima ad un problema di programmazione lineare esaminando una sequenza di punti in una regione ammissibile, cioè la regione nello spazio n-dimensionale che soddisfa $Ax \le b$. L'algoritmo è basato sul fatto che una soluzione che massimizzi la funzione obiettivo nella regione ammissibile deve occorrere ad un qualche "punto estremo" o "vertice" della regione ammissibile stessa. L'algoritmo del simplesso procede da uno spigolo all'altro della regione ammissibile finché la funzione obiettivo non può essere più migliorata. Un "simplesso" è un involucro convesso (si veda il paragrafo 35.3) di $d + 1$ punti nello spazio d-dimensionale (come un triangolo nel piano o un tetraedro nello spazio di dimensione 3). Secondo Dantzig [53], è possibile vedere l'operazione di spostarsi da un vertice ad un altro come un'operazione su di un simplesso derivata da un'interpretazione "duale" del problema di programmazione lineare; da qui il nome di "metodo del simplesso".

esponenziale. I programmi lineari generali possono essere risolti in tempo polinomiale con l'***algoritmo dell'ellissoide***, che è però lento nella pratica, oppure con l'***algoritmo di Karmarkar*** che nella pratica è spesso competitivo con il metodo del simplesso.

Questo libro non copre gli algoritmi generali di programmazione lineare a causa della complessità delle nozioni matematiche necessarie per comprenderli ed analizzarli. Tuttavia, per diversi motivi, è importante capire la strutturazione dei problemi di programmazione lineare. Primo, perché sapere che un dato problema può essere formulato come un problema di programmazione lineare di dimensione polinomiale immediatamente implica che vi è un algoritmo polinomiale per il problema. Secondo, perché vi sono diversi casi speciali di programmazione lineare per cui esistono algoritmi più veloci: per esempio, come mostrato in questo paragrafo, il problema di cammini minimi con sorgente singola è un caso speciale di programmazione lineare. Altri problemi che possono essere formulati come programmazione lineare comprendono il problema di cammino minimo per una coppia di nodi (Esercizio 25.5-4) ed il problema di flusso massimo (Esercizio 27.1-8).

A volte non si è espressamente interessati alla funzione obiettivo, ma si vuole solo trovare una qualunque ***soluzione ammissibile***, cioè un vettore x che soddisfi $Ax \le b$, oppure determinare che non esiste alcuna soluzione ammissibile. Ci concentreremo su uno di questi ***problemi di ammissibilità***.

Sistemi di vincoli di differenza

In un ***sistema di vincoli di differenza***, ogni riga della matrice di programmazione lineare A contiene un 1 ed un -1, mentre tutti gli altri elementi sono 0. Quindi, i vincoli dati da $Ax \le b$ sono un insieme di m ***vincoli di differenza*** in n incognite, in cui ogni vincolo è una semplice disuguaglianza lineare della forma

$$x_j - x_i \le b_k ,$$

dove $1 \le i, j \le n$ e $1 \le k \le m$.

Ad esempio, si consideri il problema di trovare il vettore di 5 elementi $x = (x_i)$ che soddisfi

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \le \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix} .$$

Questo problema è equivalente a trovare le incognite x_i, per $i = 1, 2, \ldots, 5$, tali che i seguenti 8 vincoli di differenza siano soddisfatti:

$$\begin{aligned} x_1 - x_2 &\le 0 , \\ x_1 - x_5 &\le -1 , \\ x_2 - x_5 &\le 1 , \\ x_3 - x_1 &\le 5 , \\ x_4 - x_1 &\le 4 , \\ x_4 - x_3 &\le -1 , \\ x_5 - x_3 &\le -3 , \\ x_5 - x_4 &\le -3 . \end{aligned} \tag{25.4}$$

Una soluzione per questo problema è $x = (-5, -3, 0, -1, -4)$, come si può verificare direttamente controllando ogni disuguaglianza. In effetti vi è più di una soluzione per questo problema: ad esempio $x' = (0, 2, 5, 4, 1)$ è un'altra soluzione. Queste due soluzioni sono correlate: ogni componente di x' è maggiore di 5 unità della corrispondente componente di x e questa non è una coincidenza.

Lemma 25.16

Sia $x = (x_1, x_2, \ldots, x_n)$ una soluzione di un sistema $Ax \le b$ di vincoli di differenza e sia d una qualunque costante. Allora $x + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$ è anch'essa una soluzione per $Ax \le b$.

Dimostrazione. Per ogni x_i e x_j, si ha $(x_j + d) - (x_i + d) = x_j - x_i$. Quindi, se x soddisfa $Ax \le b$, anche $x + d$ lo soddisfa. ■

I sistemi di vincoli di differenza compaiono in diverse applicazioni. Ad esempio, le incognite x possono essere istanti di tempo in cui certi eventi devono occorrere: allora ogni vincolo può stabilire che un certo evento non può occorrere troppo tempo dopo un altro evento. Ad esempio, gli eventi possono rappresentare lavori che devono essere eseguiti durante la costruzione di una casa. Se lo scavo delle fondamenta comincia al tempo x_1 e richiede 3 giorni, e la colata di calcestruzzo per le fondamenta comincia al tempo x_2, è ragionevole richiedere che $x_2 \le x_1 + 3$, oppure, equivalentemente, che $x_1 - x_2 \le -3$. Quindi, i vincoli di temporizzazione relativa tra gli eventi possono essere espressi come vincoli di differenza.

Grafi di vincoli

È vantaggioso interpretare i sistemi di vincoli di differenza dal punto di vista della teoria dei grafi. L'idea è che in un sistema $Ax \le b$ di vincoli di differenza, la matrice $n \times m$ di programmazione lineare A può essere vista come una matrice di incidenza (si veda l'Esercizio 23.1-7) per un grafo con n vertici ed m archi. Ogni vertice v_i del grafo, per $i = 1, 2, \ldots, n$, corrisponde ad una delle n variabili incognite x_i. Ogni arco orientato nel grafo corrisponde ad una delle m disuguaglianze che coinvolgono due incognite.

Figura 25.9 *Il grafo di vincoli corrispondente al sistema di vincoli di differenza (25.4). Il valore di $\delta(v_0, v_i)$ è indicato in ogni vertice v_i. Una soluzione ammissibile per il sistema è $x = (-5, -3, 0, -1, -4)$.*

Più formalmente, dato un sistema $Ax \le b$ di vincoli di differenza, il corrispondente ***grafo di vincoli*** è un grafo orientato e pesato $G = (V, E)$, in cui

$$V = \{v_0, v_1, \ldots, v_n\}$$

e

$$E = \{(v_i, v_j) : x_j - x_i \le b_k \quad \text{è un vincolo}\} \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \ldots, (v_0, v_n)\} .$$

Il vertice addizionale v_0 viene introdotto, come vedremo tra poco, per garantire che ogni altro vertice sia raggiungibile da esso. Quindi, l'insieme dei vertici V contiene un vertice v_i per ogni incognita x_i, più un vertice addizionale v_0. L'insieme degli archi E contiene un arco per ogni vincolo di differenza, più un arco (v_0, v_i) per ogni incognita x_i. Se $x_j - x_i \le b_k$ è un vincolo di differenza, allora il peso dell'arco (v_i, v_j) è $w(v_i, v_j) = b_k$; invece, il peso di ogni arco uscente da v_0 è 0. La figura 25.9 mostra il grafo di vincoli per il sistema di vincoli di differenza (25.4).

Il teorema seguente mostra che una soluzione di un sistema di vincoli di differenza può essere ottenuta trovando i pesi di cammino minimo nel corrispondente grafo di vincoli.

Teorema 25.17

Dato un sistema di vincoli di differenza $Ax \le b$, sia $G = (V, E)$ il corrispondente grafo di vincoli. Se G non contiene cicli di peso negativo, allora

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \tag{25.5}$$

è una soluzione ammissibile per il sistema. Se G contiene un ciclo di peso negativo, allora non vi sono soluzioni ammissibili per il sistema.

Dimostrazione. Dapprima mostriamo che se il grafo di vincoli non contiene cicli di peso negativo, allora l'equazione (25.5) fornisce una soluzione ammissibile. Si consideri un qualunque arco $(v_i, v_j) \in E$. Per il lemma 25.3, $\delta(v_0, v_j) \le \delta(v_0, v_i) + w(v_i, v_j)$ oppure, equivalentemente, $\delta(v_0, v_j) - \delta(v_0, v_i) \le w(v_i, v_j)$. Quindi, ponendo $x_j = \delta(v_0, v_j)$ e $x_i = \delta(v_0, v_i)$, il vincolo di differenza $x_j - x_i \le w(v_i, v_j)$ corrispondente all'arco (v_i, v_j) è soddisfatto.

Ora mostriamo che se il grafo di vincoli contiene un ciclo di peso negativo, allora il sistema di vincoli di differenza non ha soluzioni ammissibili. Senza perdita di generalità, si supponga che il ciclo di peso negativo sia $c = \langle v_1, v_2, \ldots, v_k \rangle$, dove $v_1 = v_k$ (il vertice v_0 non può essere sul ciclo, perché non ha archi entranti). Il ciclo c corrisponde ai seguenti vincoli di differenza:

$$\begin{aligned}
x_2 - x_1 &\le w(v_1, v_2) , \\
x_3 - x_2 &\le w(v_2, v_3) , \\
&\vdots \\
x_k - x_{k-1} &\le w(v_{k-1}, v_k) , \\
x_1 - x_k &\le w(v_k, v_1) .
\end{aligned}$$

Poiché ogni soluzione per x deve soddisfare ognuna di queste k disuguaglianze, qualunque soluzione deve anche soddisfare la disuguaglianza che si ottiene sommandole tutte insieme. Se si sommano i membri sinistri, ogni incognita x_i viene addizionata una volta e sottratta una volta, e quindi la somma delle parti sinistre è 0. La somma dei membri destri è invece $w(c)$ e quindi si ottiene $0 \le w(c)$. Ma siccome c è un ciclo di peso negativo, $w(c) < 0$ e quindi ogni soluzione per le x deve soddisfare $0 \le w(c) < 0$, il che è impossibile. ■

Risoluzione di sistemi di vincoli di differenza

Il Teorema 25.17 ci dice che è possibile utilizzare l'algoritmo di Bellman-Ford per risolvere un sistema di vincoli di differenza. Poiché vi sono archi dal vertice sorgente v_0 a tutti gli altri vertici nel grafo di vincoli, ogni eventuale ciclo di peso negativo nel grafo di vincoli è raggiungibile da v_0. Quindi se l'algoritmo di Bellman-Ford restituisce TRUE, allora i pesi di cammino minimo forniscono una soluzione ammissibile per il sistema. Nella figura 25.9, ad esempio, i pesi di cammino minimo forniscono la soluzione ammissibile $x = (-5, -3, 0, -1, -4)$ e, per il lemma 25.16, $x = (d-5, d-3, d, d-1, d-4)$ è anch'essa una soluzione ammissibile per ogni costante d. Se l'algoritmo di Bellman-Ford restituisce invece FALSE, allora non vi è alcuna soluzione ammissibile per il sistema di vincoli di differenza.

Un sistema di vincoli di differenza con m vincoli in n incognite produce un grafo con $n + 1$ vertici ed $n + m$ archi. Quindi, usando l'algoritmo di Bellman-Ford si può risolvere il sistema in tempo $O((n+1)(n+m)) = O(n^2 + nm)$. L'Esercizio 25.5-5 chiede di mostrare che l'algoritmo richiede in effetti tempo $O(nm)$, anche se m è molto minore di n.

Esercizi

25.5-1 Si trovi una soluzione ammissibile o si determini che non esistono soluzioni ammissibili per il seguente sistema di vincoli di differenza:

$$\begin{aligned}
x_1 - x_2 &\le 1 , \\
x_1 - x_4 &\le -4 , \\
x_2 - x_3 &\le 2 , \\
x_2 - x_5 &\le 7 , \\
x_2 - x_6 &\le 5 ,
\end{aligned}$$

$$
\begin{aligned}
x_3 - x_6 &\le 10, \\
x_4 - x_2 &\le 2, \\
x_5 - x_1 &\le -1, \\
x_5 - x_4 &\le 3, \\
x_6 - x_3 &\le -8.
\end{aligned}
$$

25.5-2 Si trovi una soluzione ammissibile o si determini che non esistono soluzioni ammissibili per il seguente sistema di vincoli di differenza:

$$
\begin{aligned}
x_1 - x_2 &\le 4, \\
x_1 - x_5 &\le 5, \\
x_2 - x_4 &\le -6, \\
x_3 - x_2 &\le 1, \\
x_4 - x_1 &\le 3, \\
x_4 - x_3 &\le 5, \\
x_4 - x_5 &\le 10, \\
x_5 - x_3 &\le -4, \\
x_5 - x_4 &\le -8.
\end{aligned}
$$

25.5-3 Può essere positivo un peso di cammino minimo dal nuovo vertice v_0 in un grafo di vincoli? Si giustifichi la risposta.

25.5-4 Si esprima il problema di cammino minimo per una coppia come programma lineare.

25.5-5 Si mostri come modificare leggermente l'algoritmo di Bellman-Ford in modo tale che quando esso viene usato per risolvere un sistema di vincoli di differenza con m disuguaglianze in n incognite, il tempo di esecuzione sia $O(nm)$.

25.5-6 Si mostri come si può risolvere un sistema di vincoli di differenza con l'esecuzione dell'algoritmo di Bellman-Ford su un grafo di vincoli senza il vertice addizionale v_0.

* *25.5-7* Sia $Ax \le b$ un sistema di m vincoli di differenza in n incognite. Si mostri che l'algoritmo di Bellman-Ford, quando viene eseguito sul corrispondente grafo di vincoli, massimizza la funzione $\sum_{i=1}^{n} x_i$ soggetta a $Ax \le b$ e $x_i \le 0$, per ogni x_i.

* *25.5-8* Si mostri che l'algoritmo di Bellman-Ford, quando viene eseguito sul grafo di vincoli di un sistema $Ax \le b$ di vincoli di differenza, minimizza la quantità $(\max\{x_i\} - \min\{x_i\})$ soggetta a $Ax \le b$. Si spieghi come questo fatto possa risultare conveniente se l'algoritmo è usato per allocare lavori di costruzione.

25.5-9 Si supponga che ogni riga della matrice A di un programma lineare $Ax \le b$ corrisponda ad un vincolo di differenza, ad un vincolo in una sola variabile della forma $x_i \le b_k$, oppure ad un vincolo in una sola variabile della forma $-x_i \le b_k$. Si mostri come si può adattare l'algoritmo di Bellman-Ford per risolvere questo tipo di sistemi di vincoli.

25.5-10 Si supponga che oltre ad un sistema di vincoli di differenza si vogliano gestire vincoli di uguaglianza della forma $x_i = x_j + b_k$. Si mostri come si può adattare l'algoritmo di Bellman-Ford per risolvere questo tipo di sistemi di vincoli.

25.5-11 Si dia un algoritmo efficiente per risolvere un sistema $Ax \le b$ di vincoli di differenza quando tutti gli elementi di b sono valori reali, e tutte le incognite x_i devono essere interi.

* *25.5-12* Si dia un algoritmo efficiente per risolvere un sistema $Ax \le b$ di vincoli di differenza quando tutti gli elementi di b sono valori reali e alcune delle incognite x_i, ma non necessariamente tutte, devono essere interi.

Problemi

25-1 Il miglioramento di Yen all'algoritmo di Bellman-Ford

Si supponga di eseguire il rilassamento degli archi in ogni passo dell'algoritmo di Bellman-Ford nel seguente ordine. Prima del primo passo, si assegna un ordine arbitrario $v_1, v_2, \ldots, v_{|V|}$ ai vertici del grafo di input $G = (V, E)$; quindi, si partiziona l'insieme degli archi E in $E_f \cup E_b$, dove $E_f = \{(v_i, v_j) \in E : i < j\}$ ed $E_b = \{(v_i, v_j) \in E : i > j\}$. Si definiscano $G_f = (V, E_f)$ e $G_b = (V, E_b)$.

a. Si dimostri che G_f è aciclico con ordinamento topologico $\langle v_1, v_2, \ldots, v_{|V|}\rangle$ e che G_b è aciclico con ordinamento topologico $\langle v_{|V|}, v_{|V|-1}, \ldots, v_1\rangle$.

Si supponga di realizzare ogni passo dell'algoritmo di Bellman-Ford nel modo seguente. Si visitano i vertici nell'ordine $v_1, v_2, \ldots, v_{|V|}$, rilassando gli archi di E_f che escono da ogni vertice; quindi, si visitano i vertici nell'ordine $v_{|V|}, v_{|V|-1}, \ldots, v_1$, rilassando gli archi di E_b che escono da ogni vertice.

b. Si dimostri che se G non contiene cicli di peso negativo raggiungibili dal vertice sorgente s, allora con questo schema dopo soli $\lceil |V|/2 \rceil$ passi sugli archi vale $d[v] = \delta(s, v)$ per tutti i vertici $v \in V$.

c. Che influenza ha questo schema sul tempo di esecuzione dell'algoritmo di Bellman-Ford?

25-2 Incapsulamento di scatole

Una scatola d-dimensionale con dimensioni $(x_1, x_2, \ldots, x_d)$ *si incapsula* in un'altra scatola di dimensioni $(y_1, y_2, \ldots, y_d)$ se esiste una permutazione π su $\{1, 2, \ldots, d\}$ tale che $x_{\pi(1)} < y_1$, $x_{\pi(2)} < y_2, \ldots, x_{\pi(d)} < y_d$.

a. Si dimostri che la relazione di incapsulamento è transitiva.

b. Si descriva un metodo efficiente per determinare se una scatola d-dimensionale si incapsula in un'altra scatola oppure no.

c. Si supponga di avere un insieme di n scatole d-dimensionali $\{B_1, B_2, \ldots, B_n\}$. Si descriva un algoritmo efficiente per determinare la sequenza di scatole più lunga $\langle B_{i_1}, B_{i_2}, \ldots, B_{i_k}\rangle$ tale che B_{i_j} si incapsuli in $B_{i_{j+1}}$, per $j = 1, 2, \ldots, k-1$. Si esprima il tempo di esecuzione dell'algoritmo proposto in termini di n e d.

25-3 *Arbitraggio*

Con ***arbitraggio*** si denota l'utilizzo delle discrepanze nei tassi di cambio delle valute per trasformare un'unità di una valuta in più di un'unità della stessa valuta. Ad esempio, si supponga che con un Dollaro U.S.A. si comprino 0.7 Sterline inglesi, che con una Sterlina inglese si comprino 9.5 Franchi francesi e che con un Franco francese si comprino 0.16 Dollari U.S.A. Allora un commerciante, cambiando valuta, può partire con un Dollaro U.S.A. e comprare $0.7 \times 9.5 \times 0.16 = 1.064$ Dollari U.S.A., ottenendo quindi un profitto del 6.4 per cento.

Si supponga di avere n valute $c_1, c_2, \ldots, c_n$ ed una tabella R di tassi di cambio di dimensione $n \times n$ tale che un'unità della valuta c_i compri $R[i, j]$ unità della valuta c_j.

a. Si dia un algoritmo efficiente per determinare se esiste oppure no una sequenza di valute $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k}\rangle$ tale che

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1.$$

Si analizzi il tempo di esecuzione dell'algoritmo proposto.

b. Si dia un algoritmo efficiente per stampare una tale sequenza, se ne esiste almeno una. Si analizzi il tempo di esecuzione dell'algoritmo proposto.

25-4 *Algoritmo incrementale di Gabow per cammini minimi con sorgente singola*

Un algoritmo ***incrementale*** risolve un problema considerando inizialmente solo il bit più significativo di ogni valore in input (ad esempio il peso di un arco); la soluzione iniziale viene poi raffinata considerando i due bit più significativi dell'input e successivamente vengono considerati bit sempre meno significativi, raffinando ad ogni passo la soluzione corrente finché non siano stati considerati tutti i bit: a questo punto, la soluzione calcolata è quella corretta.

In questo problema esamineremo un algoritmo per calcolare i cammini minimi da una sorgente singola, considerando incrementalmente i pesi degli archi. Sia dato un grafo orientato $G = (V, E)$ con pesi interi e non negativi w e sia $W = \max_{(u,v) \in E}\{w(u, v)\}$. Il nostro scopo è di sviluppare un algoritmo che richieda tempo $O(E \lg W)$. Assumiamo che tutti i vertici siano raggiungibili dalla sorgente.

L'algoritmo considera i bit della rappresentazione binaria dei pesi degli archi uno alla volta, dal più significativo al meno significativo. Più precisamente, sia $k = \lceil \lg(W + 1)\rceil$ il numero di bit nella rappresentazione binaria di W e, per $i = 1, 2, \ldots, k$, sia $w_i(u, v) = \lfloor w(u, v)/2^{k-i}\rfloor$. Quindi, $w_i(u, v)$ è la versione "ridotta" di $w(u, v)$, data dagli i bit più significativi di $w(u, v)$: si osservi che $w_k(u, v) = w(u, v)$ per tutti gli archi $(u, v) \in E$. Ad esempio, se $k = 5$ e $w(u, v) = 25$,

che ha come rappresentazione binaria $\langle 11001\rangle$, allora $w_3(u, v) = \langle 110\rangle = 6$; se invece $w(u, v) = \langle 00100\rangle = 4$ e k è sempre 5, allora $w_3(u, v) = \langle 001\rangle = 1$. Si definisca $\delta_i(u, v)$ come il peso di cammino minimo dal vertice u al vertice v usando la funzione peso w_i (quindi $\delta_k(u, v) = \delta(u, v)$, per tutti gli $u, v \in V$). Per un dato vertice sorgente s, l'algoritmo incrementale dapprima calcola i pesi di cammino minimo $\delta_1(s, v)$ per tutti i $v \in V$, quindi calcola $\delta_2(s, v)$ per tutti i $v \in V$ e così via, finché non calcola $\delta_k(s, v)$ per tutti i $v \in V$. Nel seguito, assumeremo che $|E| \geq |V| - 1$; vedremo che per calcolare δ_i da δ_{i-1} occorre tempo $O(E)$: quindi l'intero algoritmo richiede tempo $O(kE) = O(E \lg W)$.

a. Si supponga che per tutti i vertici $v \in V$ si abbia $\delta(s, v) \leq |E|$. Si mostri che è possibile calcolare $\delta(s, v)$, per tutti i $v \in V$, in tempo $O(E)$.

b. Si dimostri che si può calcolare $\delta_1(s, v)$, per tutti i $v \in V$, in tempo $O(E)$.

Concentriamoci ora sul calcolo di δ_i a partire da δ_{i-1}.

c. Si dimostri che, per $i = 2, 3, \ldots, k$, o vale $w_i(u, v) = 2w_{i-1}(u, v)$, oppure $w_i(u, v) = 2w_{i-1}(u, v) + 1$. Quindi, si dimostri che per tutti i $v \in V$ vale

$$2\delta_{i-1}(s, v) \leq \delta_i(s, v) \leq 2\delta_{i-1}(s, v) + |V| - 1 .$$

d. Per $i = 2, 3, \ldots, k$ e per tutti gli $(u, v) \in E$, si definisca

$$\widehat{w}_i(u, v) = w_i(u, v) + 2\delta_{i-1}(s, u) - 2\delta_{i-1}(s, v) .$$

Si dimostri che, per $i = 2, 3, \ldots, k$ e per tutti gli $u, v \in V$, il nuovo peso $\widehat{w}_i(u, v)$ dell'arco (u, v) è un intero non negativo.

e. Si definisca $\widehat{\delta}_i(s, v)$ come il peso di cammino minimo da s a v usando la funzione peso $\widehat{w}_i$. Si dimostri che, per $i = 2, 3, \ldots, k$ e per tutti i $v \in V$, vale che

$$\delta_i(s, v) = \widehat{\delta}_i(s, v) + 2\delta_{i-1}(s, v)$$

e che $\quad \widehat{\delta}_i(s, v) \leq |E|$.

f. Si mostri come si può calcolare $\delta_i(s, v)$ da $\delta_{i-1}(s, v)$, per tutti i $v \in V$, in tempo $O(E)$ e si deduca che $\delta(s, v)$ può essere calcolato, per tutti i $v \in V$, in tempo $O(E \lg W)$.

25-5 *Algoritmo di Karp per il ciclo di peso medio minimo*

Sia $G = (V, E)$ un grafo orientato con funzione peso $w: E \to \mathbf{R}$ e sia $n = |V|$. Si definisce il ***peso medio*** di un ciclo $c = \langle e_1, e_2, \ldots, e_k\rangle$ di archi in E come

$$\mu(c) = \frac{1}{k}\sum_{i=1}^{k} w(e_i) .$$

Sia $\mu^* = \min_c \mu(c)$, dove c varia su tutti i cicli orientati in G; un ciclo c per cui $\mu(c) = \mu^*$ è chiamato un ***ciclo di peso medio minimo***. In questo problema analizzeremo un algoritmo efficiente per calcolare μ^*.

Si assuma, senza perdita di generalità, che ogni vertice $v \in V$ sia raggiungibile da un vertice sorgente $s \in V$. Sia $\delta(s, v)$ il peso di un cammino minimo da s a v e sia $\delta_k(s, v)$ il peso di un cammino minimo da s a v avente *esattamente* k archi. Se non vi è nessun cammino da s a v con esattamente k archi, allora sia $\delta_k(s, v) = \infty$.

a. Si mostri che se $\mu^* = 0$, allora G non contiene cicli di peso negativo e
$\delta(s, v) = \min_{0 \le k \le n-1} \delta_k(s, v)$,
per tutti i vertici $v \in V$.

b. Si mostri che se $\mu^* = 0$, allora

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} \ge 0$$

per tutti i vertici $v \in V$. (*Suggerimento*: si usino entrambe le proprietà della parte (a).)

c. Sia c un ciclo di peso 0 e siano u e v due vertici arbitrari su c. Si supponga che il peso del cammino da u a v lungo il ciclo sia x. Si dimostri che $\delta(s, v) = \delta(s, u) + x$. (*Suggerimento*: il peso del cammino da v a u lungo il ciclo è $-x$).

d. Si mostri che se $\mu^* = 0$ allora esiste un vertice v sul ciclo di peso medio minimo tale che

$$\max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0 .$$

(*Suggerimento*: si mostri che un cammino minimo che termina ad un qualunque vertice sul ciclo di peso medio minimo può essere esteso lungo il ciclo per formare un cammino minimo che termina nel successivo vertice del ciclo.)

e. Si mostri che se $\mu^* = 0$, allora

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} = 0 .$$

f. Si mostri che se si aggiunge una costante t al peso di ogni arco di G, allora μ^* viene incrementato di t. Si usi questo fatto per mostrare che

$$\mu^* = \min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(s, v) - \delta_k(s, v)}{n - k} .$$

g. Si fornisca un algoritmo di tempo $O(VE)$ per calcolare μ^*.

Note al capitolo

L'algoritmo di Dijkstra [55] è apparso nel 1959, ma non menzionava una coda con priorità. L'algoritmo di Bellman-Ford è basato su due algoritmi distinti di Bellman [22] e Ford [71]. Bellman descrive la relazione tra cammini minimi e vincoli di differenza. Lawler [132] descrive l'algoritmo lineare per i cammini minimi in un dag che egli considera parte del folklore.

Quando i pesi degli archi sono interi relativamente piccoli, si possono usare algoritmi più efficienti per risolvere il problema di cammini minimi con sorgente singola. Ahuja, Mehlhorn, Orlin e Tarjan [6] danno un algoritmo che richiede tempo $O(E + \sqrt{\lg W})$ su grafi in cui gli archi hanno pesi non negativi, dove W è il massimo tra i pesi degli archi del grafo; essi danno anche un algoritmo facilmente programmabile che richiede tempo $O(E + V \lg W)$. Per grafi che contengono archi con peso negativo, l'algoritmo di Gabow e Tarjan [77] richiede tempo $O(\sqrt{\lg V}\, E \lg(VW))$, dove W è il massimo dei pesi degli archi del grafo.

Papadimitriou e Steiglitz [154] presentano una buona discussione del metodo del simplesso e dell'algoritmo dell'ellissoide, come anche di altri algoritmi connessi alla programmazione lineare. L'algoritmo del simplesso per la programmazione lineare fu inventato da G. Danzig nel 1947; varianti del simplesso rimangono a tutt'oggi i metodi più popolari per risolvere problemi di programmazione lineare. L'algoritmo dell'ellissoide è dovuto a L. G. Khachian nel 1979, ed è basato su di un lavoro precedente di N. Z. Shor, D. B. Judin e A. S. Nemirovskii. Karmarkar descrive il proprio algoritmo in [115].

Cammini minimi tra tutte le coppie

26

In questo capitolo consideriamo il problema di trovare i cammini minimi tra tutte le coppie di vertici in un grafo. Questo problema può sorgere per esempio nella compilazione di una tabella di distanze tra tutte le coppie di città di un atlante stradale. Come nel Capitolo 25, viene dato un grafo orientato e pesato $G = (V, E)$ con una funzione peso $W : E \rightarrow \mathbf{R}$ che associa ad ogni arco un peso a valore nei reali. Si vuole trovare, per ogni coppia di vertici $u, v \in V$, un cammino minimo (cioè di peso minimo) da u a v, dove il peso di un cammino è la somma dei pesi degli archi che lo compongono. Tipicamente, si vuole l'output in forma di tabella, in cui l'elemento nella riga u-esima e nella colonna v-esima sia il peso di un cammino minimo da u a v.

Si può risolvere un problema di cammini minimi tra tutte le coppie eseguendo un algoritmo di cammini minimi con sorgente singola $|V|$ volte, una per ogni vertice come sorgente. Se tutti i pesi degli archi sono non negativi, si può usare l'algoritmo di Dijkstra: se si realizza la coda con priorità con un array lineare, il tempo di esecuzione è $O(V^3 + VE) = O(V^3)$. La realizzazione della coda con priorità con uno heap binario permette di ottenere un tempo di esecuzione $O(VE \lg V)$ che è un miglioramento se il grafo è sparso. Altrimenti, si può realizzare la coda con priorità con uno heap di Fibonacci, ottenendo un tempo di esecuzione $O(V^2 + VE)$.

Se si permettono archi con peso negativo, l'algoritmo di Dijkstra non può essere usato e bisogna eseguire, invece, l'algoritmo più lento di Bellman-Ford una volta per ogni vertice. Il tempo di esecuzione risultante è allora $O(V^2 E)$ che su un grafo denso è $O(V^4)$. In questo capitolo vedremo come si può fare meglio; inoltre, analizzeremo la relazione tra il problema di cammini minimi tra tutte le coppie e la moltiplicazione di matrici e ne studieremo la struttura algebrica.

A differenza degli algoritmi per sorgente singola, che assumono una rappresentazione del grafo con liste di adiacenza, la maggior parte degli algoritmo di questo capitolo usa una rappresentazione con matrice di adiacenza (sebbene l'algoritmo di Johnson per grafi sparsi usi liste di adiacenza). L'input è una matrice $n \times n$ W che rappresenta i pesi degli archi di un grafo orientato $G = (V, E)$ avente n vertici. In altre parole, $W = (w_{ij})$, dove

$$w_{ij} = \begin{cases} 0 & \text{se } i = j \,, \\ \text{il peso dell'arco orientato } (i, j) & \text{se } i \neq j \quad \text{e} \quad (i, j) \in E \,, \\ \infty & \text{se } i \neq j \quad \text{e} \quad (i, j) \notin E \,. \end{cases} \tag{26.1}$$

Sono ammessi archi di peso negativo, ma per il momento assumiamo che il grafo non contenga cicli di peso negativo.

L'output tabulare degli algoritmi di cammini minimi tra tutte le coppie presentati in questo capitolo è una matrice $D = (d_{ij})$ di dimensione $n \times n$, in cui l'elemento d_{ij} contiene il peso di un cammino minimo dal vertice i al vertice j. Quindi se $\delta(i, j)$ denota il peso di cammino minimo dal vertice i al vertice j (come nel Capitolo 25), allora al termine si ha $d_{ij} = \delta(i, j)$.

Per risolvere il problema di cammini minimi tra tutte le coppie su di una matrice di adiacenza di input, si deve calcolare non solo i pesi di cammino minimo, ma anche una ***matrice dei predecessori*** $\Pi = (\pi_{ij})$, dove π_{ij} è NIL se $i = j$ oppure se non vi è alcun cammino da i a j, altrimenti π_{ij} è il predecessore di j su un cammino minimo da i. Proprio come il sottografo dei predecessori G_π del Capitolo 25 è un albero di cammini minimi per un dato vertice sorgente, così il sottografo indotto dalla i-esima riga della matrice Π sarà un albero di cammini minimi con radice i. Per ogni vertice $i \in V$, si definisce il ***sottografo dei predecessori*** di G per i come $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, dove

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\}$$

e

$$E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} \text{ e } \pi_{ij} \neq \text{NIL}\} .$$

Se $G_{\pi,i}$ è un albero di cammini minimi, allora la seguente procedura, che è una versione modificata della procedura PRINT-PATH del Capitolo 23, stampa un cammino minimo dal vertice i al vertice j.

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
if i = j
    then stampa i
    else if π_ij = NIL
            then stampa "non esiste alcun cammino da" i "a" j""
            else PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, π_ij)
                 stampa j
```

Allo scopo di mettere in evidenza le caratteristiche essenziali degli algoritmi di cammini minimi tra tutte le coppie, in questo capitolo non tratteremo la creazione e le proprietà delle matrici di predecessori così approfonditamente come abbiamo trattato quelle dei sottografi dei predecessori nel Capitolo 25. Le nozioni di base sono comunque coperte da alcuni esercizi.

Struttura del capitolo

Il paragrafo 26.1 presenta un algoritmo di programmazione dinamica basato sulla moltiplicazione di matrici per risolvere il problema di cammini minimi tra tutte le coppie. Usando la tecnica della "quadratura ripetuta", questo algoritmo richiede tempo $\Theta(V^3 \lg V)$. Un altro algoritmo di programmazione dinamica, l'algoritmo di Floyd-Warshall, viene discusso nel paragrafo 26.2: questo algoritmo richiede invece tempo $\Theta(V^3)$. Il paragrafo 26.2 considera anche il problema di trovare la chiusura transitiva di un grafo orientato, che è correlato al problema di cammini minimi tra tutte le coppie. L'algoritmo di Johnson è presentato nel

paragrafo 26.3; a differenza degli altri algoritmi di questo capitolo, quello di Johnson usa la rappresentazione con liste di adiacenza di un grafo. Esso, risolve il problema di cammini minimi tra tutte le coppie in tempo $O(V^2 \lg V + VE)$, ed è quindi un buon algoritmo per grafi grandi e sparsi. Infine, nel paragrafo 26.4 si esaminerà una struttura algebrica chiamata "semianello chiuso" che consente di applicare molti algoritmi per cammini minimi ad una moltitudine di altri problemi che coinvolgono tutte le coppie di vertici di un grafo, anche se apparentemente non hanno nulla a che vedere con cammini minimi.

Prima di procedere, dobbiamo stabilire alcune convenzioni per la rappresentazione delle matrici di adiacenza. Per prima cosa, in generale assumeremo che il grafo di input $G = (V, E)$ abbia n vertici, quindi $n = |V|$. Inoltre, useremo la convenzione di denotare le matrici con lettere maiuscole, come W o D, e di loro elementi con lettere minuscole, come w_{ij} o d_{ij}. Alcune matrici, usate in iterazioni, avranno un indice in parentesi, come in $D^{(m)} = (d_{ij}^{(m)})$. Infine, per una data matrice A di dimensione $n \times n$ assumeremo che il valore di n sia memorizzato nell'attributo *rows*[A].

26.1 Cammini minimi e moltiplicazione di matrici

Questo paragrafo presenta un algoritmo di programmazione dinamica per il problema di cammini minimi tra tutte le coppie per un grafo orientato $G = (V, E)$. Il ciclo principale del programma dinamico invocherà un'operazione che è molto simile alla moltiplicazione di matrici, quindi l'algoritmo avrà l'aspetto di una moltiplicazione iterata di matrici. Inizieremo sviluppando un algoritmo di tempo $\Theta(V^4)$ e quindi ne miglioreremo il tempo di esecuzione a $\Theta(V^3 \lg V)$.

Prima di procedere, riepiloghiamo brevemente i passi necessari per sviluppare un algoritmo di programmazione dinamica, così come sono stati presentati nel Capitolo 16.

1. Caratterizzare la struttura di una soluzione ottima.
2. Definire ricorsivamente il valore di una soluzione ottima.
3. Calcolare il valore di una soluzione ottima in maniera bottom-up.

(Il quarto passo, che consiste nel costruire una soluzione ottima dalle informazioni calcolate, verrà trattato negli esercizi.)

Struttura di un cammino minimo

Iniziamo col caratterizzare la struttura di una soluzione ottima. Per il problema di cammini minimi tra tutte le coppie per un grafo $G = (V, E)$, abbiamo dimostrato (nel lemma 25.1) che tutti i sottocammini di un cammino minimo sono anch'essi cammini minimi. Si supponga che il grafo sia rappresentato da una matrice di adiacenza $W = (w_{ij})$. Si consideri un cammino minimo p dal vertice i al vertice j e si supponga che p contenga al massimo m archi; assumendo che non vi siano cicli di peso negativo, m è finito. Se $i = j$, allora p ha peso 0 e non ha nessun arco; se invece i vertici i e j sono distinti, allora si può decomporre il cammino p in $i \xrightarrow{p'} k \to j$, dove il cammino p' contiene al massimo $m - 1$ archi. Inoltre, per il lemma 25.1, p' è un cammino minimo da i a k e quindi, per il corollario 25.2, abbiamo $\delta(i, j) = \delta(i, k) + w_{kj}$.

Una soluzione ricorsiva per il problema di cammini minimi tra tutte le coppie

Sia $d_{ij}^{(m)}$ il minimo tra i pesi dei cammini dal vertice i al vertice j che contengono al più m archi. Quando $m = 0$, vi è un cammino minimo da i a j senza archi se e solo se $i = j$; quindi

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{se } i = j\,, \\ \infty & \text{se } i \neq j\,. \end{cases}$$

Per $m \geq 1$, possiamo calcolare $d_{ij}^{(m)}$ come il minimo tra $d_{ij}^{(m-1)}$ (il peso del cammino minimo da i a j costituito da al più $m-1$ archi) ed il minimo tra i pesi dei cammini da i a j composti da al più m archi, ottenuto considerando tutti i possibili predecessori k di j. Quindi, definiamo ricorsivamente

$$\begin{aligned} d_{ij}^{(m)} &= \min\left(d_{ij}^{(m-1)}, \min_{1 \leq k \leq n}\left\{d_{ik}^{(m-1)} + w_{kj}\right\}\right) \\ &= \min_{1 \leq k \leq n}\left\{d_{ik}^{(m-1)} + w_{kj}\right\}\,. \end{aligned} \tag{26.2}$$

L'ultima uguaglianza segue dal fatto che $w_{jj} = 0$, per ogni j.

Quali sono i reali pesi di cammino minimo $\delta(i, j)$? Se il grafo non contiene cicli di peso negativo, allora tutti i cammini minimi sono semplici e quindi contengono al più $n-1$ archi: un cammino dal vertice i al vertice j con più di $n-1$ archi non può avere un peso minore di un cammino minimo da i a j. Quindi i pesi di cammino minimo effettivi sono dati da

$$\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \cdots\,. \tag{26.3}$$

Calcolo bottom-up dei pesi di cammino minimo

Prendendo come input la matrice $W = (w_{ij})$, calcoleremo ora una sequenza di matrici $D^{(1)}, D^{(2)}, \ldots, D^{(n-1)}$ dove, per $m = 1, 2, \ldots, n-1$, si ha $D^{(m)} = (d_{ij}^{(m)})$. La matrice finale $D^{(n-1)}$ contiene i pesi di cammino minimo effettivi. Si osservi che, poiché $d_{ij}^{(1)} = w_{ij}$ per tutti i vertici $i, j \in V$, abbiamo $D^{(1)} = W$.

Il cuore dell'algoritmo è la seguente procedura che, date le matrici $D^{(m-1)}$ e W, restituisce la matrice $D^{(m)}$: essa estende di un arco i cammini minimi calcolati sino a quel momento.

```
EXTEND-SHORTEST-PATHS(D, W)
n ← rows[D]
sia D' = (d'_ij) una matrice n × n
for i ← 1 to n
    do for j ← 1 to n
        do d'_ij ← ∞
            for k ← 1 to n
                do d'_ij ← min(d'_ij, d_ik + w_kj)
return D'
```


La procedura calcola una matrice $D' = (d'_{ij})$ che viene restituita come output. Ciò viene fatto calcolando l'equazione (26.2) per ogni i e j, usando D per $D^{(m-1)}$ e D' per $D^{(m)}$ (le matrici sono scritte senza gli indici per rendere l'input e l'output indipendenti da m). Il tempo di esecuzione è $\Theta(n^3)$ a causa dei tre cicli **for** annidati.

Possiamo ora esaminare la relazione con la moltiplicazione di matrici. Si supponga di voler calcolare il prodotto matriciale $C = A \cdot B$ di due matrici A e B di dimensione $n \times n$. Allora per $i, j = 1, 2, \ldots, n$, calcoliamo

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}\,. \tag{26.4}$$

Si osservi che se applichiamo nell'equazione (26.2) le sostituzioni

$$\begin{aligned} d^{(m-1)} &\rightarrow a\,, \\ w &\rightarrow b\,, \\ d^{(m)} &\rightarrow c\,, \\ \min &\rightarrow +\,, \\ + &\rightarrow \cdot \end{aligned}$$

otteniamo proprio l'equazione (26.4). Quindi se applichiamo questi cambiamenti alla procedura EXTEND-SHORTEST-PATHS e sostituiamo anche ∞ (l'identità per min) con 0 (l'identità per +), otteniamo l'usuale procedura per la moltiplicazione di matrici di tempo $\Theta(n^3)$.

```
MATRIX-MULTIPLY(A, B)
n ← rows[A]
sia C una matrice n × n
for i ← 1 to n
    do for j ← 1 to n
        do c_ij ← 0
            for k ← 1 to n
                do c_ij ← c_ij + a_ik · b_kj
return C
```

Tornando al problema di cammini minimi tra tutte le coppie, si calcolano i pesi di cammino minimo estendendo i cammini minimi di un arco alla volta. Se indichiamo con $A \cdot B$ la matrice "prodotto" restituita da EXTEND-SHORTEST-PATHS(A, B), calcoliamo la sequenza di $n-1$ matrici

$$\begin{aligned} D^{(1)} &= D^{(0)} \cdot W &&= W^1\,, \\ D^{(2)} &= D^{(1)} \cdot W &&= W^2\,, \\ D^{(3)} &= D^{(2)} \cdot W &&= W^3\,, \\ &\vdots \\ D^{(n-1)} &= D^{(n-2)} \cdot W &&= W^{n-1}\,. \end{aligned}$$

Come discusso precedentemente, la matrice $D^{(n-1)} = W^{n-1}$ contiene i pesi di cammino minimo. La seguente procedura calcola questa sequenza in tempo $\Theta(n^4)$.

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad D^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Figura 26.1 *Un grafo orientato e la sequenza delle matrici $D^{(m)}$ calcolate da* SLOW-ALL-PAIRS-SHORTEST-PATHS. *Il lettore può verificare che $D^{(5)} = D^{(4)} \cdot W$ è uguale a $D^{(4)}$ e quindi $D^{(m)} = D^{(4)}$ per tutti gli $m \geq 4$.*

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  D^(1) ← W
3  for m ← 2 to n − 1
4      do D^(m) ← EXTEND-SHORTEST-PATHS(D^(m−1), W)
5  return D^(n−1)
```

La figura 26.1 mostra un grafo e le matrici $D^{(m)}$ calcolate dalla procedura SLOW-ALL-PAIRS-SHORTEST-PATHS.

Miglioramento del tempo di esecuzione

Il nostro scopo, tuttavia, non è di calcolare *tutte* le matrici $D^{(m)}$: siamo interessati solo alla matrice $D^{(n-1)}$. Si ricordi che in assenza di cicli di peso negativo, l'equazione (26.3) implica che $D^{(m)} = D^{(n-1)}$ per tutti gli interi $m \geq n - 1$. Quindi, possiamo calcolare $D^{(n-1)}$ con soli $\lceil \lg(n-1) \rceil$ prodotti di matrici, calcolando la sequenza

$$\begin{aligned} D^{(1)} &= W, \\ D^{(2)} &= W^2 &&= W \cdot W, \\ D^{(4)} &= W^4 &&= W^2 \cdot W^2 \\ D^{(8)} &= W^8 &&= W^4 \cdot W^4, \\ &\vdots \\ D^{(2^{\lceil \lg(n-1) \rceil})} &= W^{2^{\lceil \lg(n-1) \rceil}} &&= W^{2^{\lceil \lg(n-1) \rceil - 1}} \cdot W^{2^{\lceil \lg(n-1) \rceil - 1}}. \end{aligned}$$

Poiché $2^{\lceil \lg(n-1) \rceil} \geq n - 1$, il prodotto finale $D^{(2^{\lceil \lg(n-1) \rceil})}$ è uguale a $D^{(n-1)}$.

La seguente procedura calcola la precedente sequenza di matrici usando questa tecnica di *quadratura ripetuta*.

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
1  n ← rows[W]
2  D^(1) ← W
3  m ← 1
4  while n − 1 > m
5      do D^(2m) ← EXTEND-SHORTEST-PATHS(D^(m), D^(m))
6         m ← 2m
7  return D^(m)
```

In ogni iterazione del ciclo **while** delle linee 4-6, si calcola $D^{(2m)} = (D^{(m)})^2$, partendo con $m = 1$. Alla fine di ogni iterazione si raddoppia il valore di m; l'ultima iterazione calcola $D^{(n-1)}$. In effetti viene calcolata $D^{(2m)}$ per qualche $n - 1 \leq 2m \leq 2n - 2$ e, per l'equazione (26.3), $D^{(2m)} = D^{(n-1)}$. Al prossimo controllo alla linea 4, m è stato raddoppiato; quindi $n - 1 \leq m$, il controllo fallisce e la procedura restituisce l'ultima matrice calcolata.

Il tempo di esecuzione della procedura FASTER-ALL-PAIRS-SHORTEST-PATHS è $\Theta(n^3 \lg n)$, poiché ognuna delle $\lceil \lg(n-1) \rceil$ matrici prodotto richiede tempo $\Theta(n^3)$. Si osservi che il codice è compatto, poiché non contiene strutture di dati elaborate e quindi la costante nascosta nella notazione Θ è piccola.

Esercizi

26.1-1 Si esegua SLOW-ALL-PAIRS-SHORTEST-PATHS sul grafo orientato e pesato di figura 26.2, mostrando le matrici che risultano ad ogni iterazione del ciclo. Quindi, si faccia lo stesso con FASTER-ALL-PAIRS-SHORTEST-PATHS.

26.1-2 Perché si richiede che $w_{ii} = 0$ per ogni $1 \leq i \leq n$?

26.1-3 A cosa corrisponde, nella normale moltiplicazione di matrici, la seguente matrice usata negli algoritmi di cammini minimi?

$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

26.1-4 Si mostri come si può esprimere il problema di cammini minimi con sorgente singola come prodotto di matrici e di un vettore. Si descriva come la valutazione

Figura 26.2 *Un grafo orientato e pesato usato negli Esercizi 26.1-1, 26.2-1 e 26.3-1.*

di questo prodotto corrisponda ad un algoritmo simile a quello di Bellman-Ford (si veda il paragrafo 25.3).

26.1-5 Si supponga di voler calcolare i vertici dei cammini minimi negli algoritmi di questo paragrafo. Si mostri come calcolare in tempo $O(n^3)$ la matrice Π dei predecessori a partire dalla matrice completa dei pesi di cammino minimo D.

26.1-6 I vertici dei cammini minimi possono anche essere calcolati contemporaneamente ai pesi di cammino minimo. Si definisca $\pi_{ij}^{(m)}$ come il predecessore del vertice j su un qualche cammino di peso minimo da i a j che contiene al massimo m archi. Si modifichino le procedure EXTEND-SHORTEST-PATHS e SLOW-ALL-PAIRS-SHORTEST-PATHS in modo da calcolare le matrici $\Pi^{(1)}, \Pi^{(2)}, \ldots, \Pi^{(n-1)}$ quando vengono calcolate le matrici $D^{(1)}, D^{(2)}, \ldots, D^{(n-1)}$.

26.1-7 La procedura FASTER-ALL-PAIRS-SHORTEST-PATHS, così come è scritta, richiede di memorizzare $\lceil \lg(n-1) \rceil$ matrici, ognuna con n^2 elementi, per una occupazione di spazio complessiva di $\Theta(n^2 \lg n)$. Si modifichi la procedura in modo che richieda solo spazio $\Theta(n^2)$, usando solo due matrici $n \times n$.

26.1-8 Si modifichi FASTER-ALL-PAIRS-SHORTEST-PATHS in modo da rilevare la presenza di un ciclo di peso negativo.

26.1-9 Si dia un algoritmo efficiente per trovare in un grafo la lunghezza (numero di archi) di un ciclo di peso negativo di lunghezza minima.

26.2 Algoritmo di Floyd-Warshall

In questo paragrafo useremo una differente formulazione di programmazione dinamica per risolvere il problema di cammini minimi tra tutte le coppie per un grafo orientato $G = (V, E)$. L'algoritmo risultante, conosciuto come ***algoritmo di Floyd-Warshall***, ha tempo d'esecuzione $\Theta(V^3)$. Come prima, vi possono essere archi di peso negativo ma assumiamo che non vi siano cicli di peso negativo. Come nel paragrafo 26.1, useremo la tecnica di programmazione dinamica per sviluppare l'algoritmo; dopo aver studiato l'algoritmo risultante, presenteremo un metodo simile per trovare la chiusura transitiva di un grafo orientato.

Figura 26.3 *Il cammino p è un cammino minimo dal vertice i al vertice j e k è il vertice intermedio di p con etichetta più grande. Il cammino p_1, cioè la porzione del cammino p dal vertice i al vertice k, ha tutti i nodi intermedi nell'insieme $\{1, 2, \ldots, k-1\}$ e lo stesso vale per il cammino p_2 dal vertice k al vertice j.*

Struttura di un cammino minimo

Nell'algoritmo di Floyd-Warshall, useremo una caratterizzazione della struttura di un cammino minimo diversa rispetto a quella usata nell'algoritmo basato sulla moltiplicazione di matrici. L'algoritmo considera i vertici "intermedi" di un cammino minimo, dove un vertice ***intermedio*** di un cammino semplice $p = \langle v_1, v_2, \ldots, v_l \rangle$ è un qualunque vertice di p diverso da v_1 e da v_l, cioè un qualunque vertice nell'insieme $\{v_2, v_3, \ldots, v_{l-1}\}$.

L'algoritmo di Floyd-Warshall è basato sulla seguente osservazione. Sia $V = \{1, 2, \ldots, n\}$ l'insieme dei vertici di G e si consideri un sottoinsieme $\{1, 2, \ldots, k\}$ di vertici, per un qualche k. Per ogni coppia di vertici $i, j \in V$, si considerino tutti i cammini da i a j i cui vertici intermedi stanno in $\{1, 2, \ldots, k\}$ e sia p un cammino di peso minimo tra di essi. (Il cammino p è semplice, poiché assumiamo che G non abbia cicli di peso negativo). L'algoritmo di Floyd-Warshall sfrutta una relazione tra il cammino p ed i cammini minimi da i a j aventi tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$. Questa relazione cambia a seconda che k sia un vertice intermedio del cammino p oppure no.

- Se k non è un vertice intermedio del cammino p, allora tutti i vertici intermedi sul cammino p sono nell'insieme $\{1, 2, \ldots, k-1\}$. Quindi, un cammino minimo dal vertice i al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$ è anche un cammino minimo da i a j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k\}$.

- Se invece k è un vertice intermedio sul cammino p, allora spezziamo p in $i \xrightarrow{p_1} k \xrightarrow{p_2} j$, come mostrato nella figura 26.3. Per il lemma 25.1, p_1 è un cammino minimo da i a k con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$; analogamente, p_2 è un cammino minimo dal vertice k al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$.

Una soluzione ricorsiva per il problema di cammini minimi tra tutte le coppie

Sulla base dell'osservazione precedente, proponiamo una nuova formulazione ricorsiva delle stime di cammino minimo, diversa da quella usata nel paragrafo 26.1. Sia $d_{ij}^{(k)}$ il peso di un cammino minimo dal vertice i al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k\}$. Quando $k = 0$, un cammino dal vertice i al vertice j senza vertici intermedi aventi un numero

maggiore di 0 è un cammino che non ha alcun vertice intermedio: di conseguenza esso ha al massimo un arco e quindi $d_{ij}^{(0)} = w_{ij}$. Una definizione ricorsiva è data da

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{se } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{se } k \geq 1\,. \end{cases} \tag{26.5}$$

La matrice $D^{(n)} = (d_{ij}^{(n)})$ fornisce la risposta finale, cioè $d_{ij}^{(n)} = \delta(i,j)$ per tutti gli $i, j \in V$, perché tutti i vertici intermedi di un qualunque cammino sono nell'insieme $\{1, 2, \ldots, n\}$.

Calcolo bottom-up dei pesi di cammino minimo

Sfruttando la ricorrenza (26.5), la seguente procedura bottom-up può essere usata per calcolare i valori $d_{ij}^{(k)}$ in ordine crescente di k. Il suo input è una matrice W di dimensione $n \times n$, definita come nell'equazione (26.1). La procedura restituisce la matrice $D^{(n)}$ dei pesi di cammino minimo.

```
FLOYD-WARSHALL(W)
1  n ← rows[W]
2  D^(0) ← W
3  for k ← 1 to n
4      do for i ← 1 to n
5          do for j ← 1 to n
6              do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D^(n)
```

La figura 26.4 mostra un grafo orientato e le relative matrici $D^{(k)}$ calcolate con l'algoritmo di Floyd-Warshall.

Il tempo di esecuzione dell'algoritmo di Floyd-Warshall è determinato dai tre cicli **for** annidati alle linee 3-6. Ogni esecuzione della linea 6 richiede tempo $O(1)$ e quindi l'algoritmo richiede tempo $\Theta(n^3)$. Come nell'ultimo algoritmo del paragrafo 26.1, il codice è compatto e senza strutture di dati elaborate, quindi la costante nascosta nella notazione Θ è piccola. Di conseguenza l'algoritmo di Floyd-Warshall è abbastanza pratico, anche per grafi di input di dimensione media.

Costruzione di un cammino minimo

Esiste una varietà di metodi per costruire i cammini minimi nell'algoritmo di Floyd-Warshall. Un modo consiste nel calcolare la matrice D dei pesi di cammino minimo e quindi di costruire la matrice dei predecessori Π dalla matrice D: ciò può essere realizzato in tempo $O(n^3)$ (Esercizio 26.1-5). Data la matrice dei predecessori Π, si può usare la procedura PRINT-ALL-PAIRS-SHORTEST-PATH per stampare i vertici lungo un dato cammino minimo.

Si può anche calcolare la matrice dei predecessori Π "in linea" mentre l'algoritmo di Floyd-Warshall calcola le matrici $D^{(k)}$. Più precisamente, si calcola una sequenza di matrici $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$, dove $\Pi = \Pi^{(n)}$ e $\pi_{ij}^{(k)}$ è definito come il predecessore del vertice j su un cammino minimo dal vertice i avente tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k\}$.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figura 26.4 *La sequenza di matrici $D^{(k)}$ e $\Pi^{(k)}$ calcolate dell'algoritmo di Floyd-Warshall per il grafo di figura 26.1.*

Si può dare una formulazione ricorsiva di $\pi_{ij}^{(k)}$. Quando $k = 0$, un cammino minimo da i a j non contiene alcun vertice intermedio e quindi

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{se } i = j \text{ oppure } w_{ij} = \infty, \\ i & \text{se } i \neq j \text{ e } w_{ij} < \infty. \end{cases} \tag{26.6}$$

Per $k \geq 1$, se prendiamo il cammino $i \leadsto k \leadsto j$, allora il predecessore di j scelto è lo stesso vertice che avevamo scelto come predecessore di j su un cammino minimo da k con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$. Altrimenti, scegliamo lo stesso predecessore

di j che avevamo scelto su un cammino minimo da i con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k-1\}$. Formalmente, per $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases} \tag{26.7}$$

Il problema di incorporare il calcolo della matrice $\Pi^{(k)}$ nella procedura FLOYD-WARSHALL verrà affrontato nell'Esercizio 26.2-3. La figura 26.4 mostra la sequenza delle matrici $\Pi^{(k)}$ generate dall'algoritmo risultante per il grafo di figura 26.1. L'esercizio propone anche il compito più difficile di dimostrare che il sottografo dei predecessori $G_{\pi,i}$ è un albero di cammini minimi con radice i. Un altro modo di ricostruire i cammini minimi è proposto come Esercizio 20.2-6.

Chiusura transitiva di un grafo orientato

Dato un grafo orientato $G = (V, E)$ con insieme di vertici $V = \{1, 2, \ldots, n\}$, ci si può chiedere se esiste un cammino in G da i a j, per ogni coppia di vertici $i, j \in V$. La ***chiusura transitiva*** di G è definita come il grafo $G^* = (V, E^*)$, dove

$E^* = \{(i, j)$: esiste un cammino in G dal vertice i al vertice $j\}$.

Un modo di calcolare la chiusura transitiva di G in tempo $\Theta(n^3)$ consiste nell'assegnare peso unitario ad ogni arco in E ed eseguire l'algoritmo di Floyd-Warshall: se vi è un cammino dal vertice i al vertice j si ottiene $d_{ij} < n$, altrimenti $d_{ij} = \infty$.

Esiste un altro modo, simile, di calcolare la chiusura transitiva di G in tempo $\Theta(n^3)$, che in pratica può far risparmiare tempo e spazio. Questo metodo consiste nel sostituire gli operatori logici $\vee$ e $\wedge$ al posto degli operatori aritmetici min e + nell'algoritmo di Floyd-Warshall. Per $i, j, k = 1, 2, \ldots, n$, sia $t_{ij}^{(k)} = 1$ se vi è un cammino nel grafo G dal vertice i al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k\}$, e $t_{ij}^{(k)} = 0$ in caso contrario. Si costruisce allora la chiusura transitiva $G^* = (V, E^*)$ mettendo l'arco (i, j) in E^* se e solo se $t_{ij}^{(n)} = 1$. Una definizione ricorsiva di $t_{ij}^{(k)}$, simile alla ricorrenza (26.5), è data da

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{se } i \neq j \text{ e } (i, j) \notin E \\ 1 & \text{se } i = j \text{ o } (i, j) \in E \end{cases}$$

e, per $k \geq 1$,

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)} \right) . \tag{26.8}$$

Come nell'algoritmodi Floyd-Warshall, possiamo calcolare le matrici $T^{(k)} = (t_{ij}^{(k)})$ in ordine crescente di k.

```
TRANSITIVE-CLOSURE(G)
n ← |V[G]|
for i ← 1 to n
    do for j ← 1 to n
        do if i = j oppure (i, j) ∈ E[G]
            then t_ij^(0) ← 1
            else t_ij^(0) ← 0
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do t_ij^(k) ← t_ij^(k-1) ∨ (t_ik^(k-1) ∧ t_kj^(k-1))
return T^(n)
```

$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figura 26.5 *Un grafo orientato e le matrici $T^{(k)}$ calcolate dall'algoritmo della chiusura transitiva.*

La figura 26.5 mostra le matrici $T^{(k)}$ calcolate dalla procedura TRANSITIVE-CLOSURE per un grafo campione. Come per l'algoritmo di Floyd-Warshall, il tempo di esecuzione della procedura TRANSITIVE-CLOSURE è $\Theta(n^3)$. Tuttavia su alcuni calcolatori le operazioni logiche su valori di un singolo bit vengono eseguite più velocemente delle operazioni aritmetiche su intere parole di memoria. Inoltre, poiché l'algoritmo di chiusura transitiva usa solo valori booleani invece di valori interi, lo spazio richiesto è minore di quello richiesto dall'algoritmo di Floyd-Warshall di un fattore corrispondente alla dimensione di una parola di memoria.

Nel paragrafo 26.4 vedremo che la corrispondenza tra FLOYD-WARSHALL e TRANSITIVE-CLOSURE è più di una coincidenza, perché entrambi gli algoritmi sono basati su un tipo di struttura algebrica chiamata "semianello chiuso".

Esercizi

26.2-1 Si esegua l'algoritmo di Floyd-Warshall sul grafo orientato e pesato di figura 26.2. Si mostri la matrice $D^{(k)}$ ad ogni iterazione del ciclo più esterno.

26.2-2 Per quanto visto precedentemente, l'algoritmo di Floyd-Warshall sembra richiedere spazio $\Theta(n^3)$, poiché viene calcolato $d_{ij}^{(k)}$, per $i, j, k = 1, 2, \ldots, n$. Si mostri che la

seguente procedura, in cui vengono semplicemente cancellati tutti gli indici, è corretta, e che quindi lo spazio effettivamente richiesto è $\Theta(n^2)$.

```
FLOYD-WARSHALL'(W)
n ← rows[W]
D ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            d_ij ← min(d_ij, d_ik + d_kj)
return D
```

26.2-3 Si modifichi la procedura FLOYD-WARSHALL in modo da comprendere il calcolo delle matrici $\Pi^{(k)}$ secondo le equazioni (26.6) e (26.7). Si dimostri rigorosamente che, per ogni $i \in V$, il grafo dei predecessori $G_{\pi,i}$ è un albero di cammini minimi con radice i. (*Suggerimento*: per mostrare che $G_{\pi,i}$ è aciclico, si mostri prima che $\pi_{ij}^{(k)} = l$ implica $d_{ij}^{(k)} \geq \pi_{il}^{(k-1)} + w_{lj}$. Quindi si adatti la dimostrazione del lemma 25.8).

26.2-4 Si supponga di modificare il modo in cui è trattata l'uguaglianza nell'equazione (26.7):

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)} , \\ \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} . \end{cases}$$

Questa definizione alternativa della matrice dei predecessori Π è corretta?

26.2-5 Come si può usare l'output dell'algoritmo di Floyd-Warshall per rilevare la presenza di un ciclo di peso negativo?

26.2-6 Un altro modo di ricostruire i cammini minimi nell'algoritmo di Floyd-Warshall consiste nell'usare i valori $\varphi_{ij}^{(k)}$, per $i, j, k = 1, 2, \ldots, n$, dove $\varphi_{ij}^{(k)}$ è il vertice intermedio più grande di un cammino minimo da i a j. Si dia una definizione ricorsiva di $\varphi_{ij}^{(k)}$, si modifichi la procedura FLOYD-WARSHALL in modo da calcolare i valori $\varphi_{ij}^{(k)}$ e si riscriva la procedura PRINT-ALL-PAIRS-SHORTEST-PATH in modo che accetti come input la matrice $\Phi = \left(\phi_{ij}^{(n)}\right)$. Qual è la relazione tra la matrice Φ e la tabella s nel problema della moltiplicazione di una catena di matrici del paragrafo 16.1?

26.2-7 Si dia un algoritmo di tempo $O(VE)$ per calcolare la chiusura transitiva di un grafo orientato $G = (V, E)$.

26.2-8 Si supponga che la chiusura transitiva di un grafo orientato e aciclico possa essere calcolata in tempo $f(V, E)$, dove $f(V, E) = \Omega(V + E)$ e f è monotona crescente. Si mostri che il tempo necessario per calcolare la chiusura transitiva di un generico grafo orientato è $O(f(V, E))$.

26.3 Algoritmo di Johnson per grafi sparsi

L'algoritmo di Johnson trova i cammini minimi tra tutte le coppie in tempo $O(V^2 \lg V + VE)$; quindi esso è asintoticamente migliore sia della quadratura ripetuta che dell'algoritmo di Floyd-Warshall per grafi sparsi. L'algoritmo restituisce una matrice di pesi di cammino minimo tra tutte le coppie, oppure denuncia la presenza di un ciclo di peso negativo nel grafo di input. L'algoritmo di Johnson usa come sottoprocedure sia l'algoritmo di Dijkstra che quello di Bellman-Ford descritti nel Capitolo 25.

L'algoritmo di Johnson usa la tecnica della ***rideterminazione dei pesi*** che funziona nel modo seguente. Se tutti i pesi degli archi w in un grafo $G = (V, E)$ sono non negativi, si possono trovare i cammini minimi tra tutte le coppie di vertici eseguendo l'algoritmo di Dijkstra una volta per ogni vertice; se la coda con priorità è realizzata tramite uno heap di Fibonacci, il tempo di esecuzione di questo algoritmo tra tutte le coppie è $O(V^2 \lg V + VE)$. Se G ha degli archi di peso negativo, si calcola semplicemente un nuovo insieme di pesi di archi non negativi che ci permette di utilizzare lo stesso metodo. Il nuovo insieme di pesi di archi w deve soddisfare due importanti proprietà.

1. Per ogni coppia di vertici $u, v \in V$, un cammino minimo da u a v secondo la funzione peso w deve essere anche un cammino minimo da u a v secondo la funzione peso $\hat{w}$.
2. Per tutti gli archi (u, v), il nuovo peso $\hat{w}(u, v)$ deve essere non negativo.

Come vedremo tra poco, la pre-elaborazione di G necessaria per determinare la nuova funzione peso $\hat{w}$ può essere eseguita in tempo $O(VE)$.

Mantenimento dei cammini minimi con la rideterminazione dei pesi

Come mostra il prossimo lemma, è facile ottenere una rideterminazione dei pesi degli archi che soddisfi la prima delle proprietà precedenti. Useremo δ per denotare i pesi di cammino minimo derivati dalla funzione peso w e $\hat{\delta}$ per denotare i pesi di cammino minimo derivati dalla funzione peso $\hat{w}$.

Lemma 26.1 (La rideterminazione dei pesi non cambia i cammini minimi)
Dato un grafo orientato e pesato $G = (V, E)$ con funzione peso $w : E \to \mathbf{R}$, sia $h : V \to \mathbf{R}$ una qualunque funzione che associa numeri reali ai vertici. Si definisca per ogni arco $(u, v) \in E$

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v). \tag{26.9}$$

Sia $p = \langle v_0, v_1, \ldots, v_k \rangle$ un cammino dal vertice v_0 al vertice v_k. Allora $w(p) = \delta(v_0, v_k)$ se e solo se $\hat{w}(p) = \hat{\delta}(v_0, v_k)$. Inoltre, G ha un ciclo di peso negativo secondo la funzione peso w se e solo se G ha un ciclo di peso negativo secondo la funzione peso $\hat{w}$.

Dimostrazione. Cominciamo col mostrare che

$$\hat{w}(p) = w(p) + h(v_0) - h(v_k) . \tag{26.10}$$

Infatti, abbiamo

$$\begin{aligned}
\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\
&= \sum_{i=1}^{k} (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
&= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
&= w(p) + h(v_0) - h(v_k) .
\end{aligned}$$

La terza uguaglianza segue dalla somma telescopica alla seconda linea della formula precedente.

Mostriamo ora, per assurdo, che $w(p) = \delta(v_0, v_k)$ implica $\hat{w}(p) = \hat{\delta}(v_0, v_k)$. Supponiamo che esista un cammino più corto p' da v_0 a v_k secondo la funzione peso $\hat{w}$, quindi che $\hat{w}(p') < \hat{w}(p)$. Per l'equazione (26.10),

$$\begin{aligned}
w(p') + h(v_0) - h(v_k) &= \widehat{w}(p') \\
&< \widehat{w}(p) \\
&= w(p) + h(v_0) - h(v_k) ,
\end{aligned}$$

che implica $w(p') < w(p)$. Ma questo contraddice l'ipotesi che p sia un cammino minimo da u a v secondo w. La dimostrazione dell'implicazione inversa è simile.

Infine, mostriamo che G ha un ciclo di peso negativo secondo la funzione peso w se e solo se G ha un ciclo di peso negativo secondo la funzione peso $\hat{w}$. Infatti, si consideri un qualunque ciclo $c = \langle v_0, v_1, \ldots, v_k \rangle$, dove $v_0 = v_k$. Per l'equazione (26.10),

$$\begin{aligned}
\widehat{w}(c) &= w(c) + h(v_0) - h(v_k) \\
&= w(c) ,
\end{aligned}$$

e quindi c ha peso negativo secondo w se e solo se ha peso negativo secondo $\hat{w}$. ■

Generazione di pesi non negativi tramite la rideterminazione dei pesi

Il prossimo obiettivo è di assicurare che valga la seconda proprietà: si vuole che $\hat{w}(u, v)$ sia non negativo per tutti gli archi $(u, v) \in E$. Dato un grafo orientato e pesato $G = (V, E)$ con funzione peso $w : E \to \mathbf{R}$, costruiamo un nuovo grafo $G' = (V', E')$, dove $V' = V \cup \{s\}$ per un qualche nuovo vertice $s \notin V$ ed $E' = E \cup \{(s, v) : v \in V\}$; inoltre la funzione peso viene estesa in modo tale che $w(s, v) = 0$ per tutti i $v \in V$. Si noti che, poiché s non ha archi entranti, nessun cammino minimo in G', ad eccezione di quelli che partono da s, può contenere s stesso. Inoltre, G' non ha cicli di peso negativo se e solo se G non ha cicli di peso negativo. La figura 26.6(a) mostra il grafo G' corrispondente al grafo G di figura 26.1.

Si supponga ora che G e G' non abbiano cicli di peso negativo e si definisca $h(v) = \delta(s, v)$ per tutti i $v \in V'$. Per il lemma 25.3, abbiamo che $h(v) \le h(u) + w(u, v)$ per tutti gli archi $(u, v) \in E'$; quindi, se definiamo i nuovi pesi $\hat{w}$ secondo l'equazione (26.9), abbiamo $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \ge 0$, e la seconda proprietà è soddisfatta. La figura 26.6(b) mostra il grafo G' della figura 26.6(a) con i nuovi pesi per gli archi.

Figura 26.6 *L'algoritmo di Johnson per i cammini minimi tra tutte le coppie eseguito sul grafo di figura 26.1.* (a) *Il grafo G' con la funzione peso originale w. Il nuovo vertice s è bianco. All'interno di ogni vertice v è indicato $h(v) = \delta(s, v)$.* (b) *Il peso di ogni arco (u, v) viene rideterminato con la funzione peso $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$.* (c)-(g) *Il risultato dell'esecuzione dell'algoritmo di Dijkstra su ogni vertice di G secondo la funzione peso $\hat{w}$. In ogni parte della figura il vertice sorgente v è nero. All'interno di ogni vertice v sono indicati i valori $\hat{\delta}(u, v)$ e $\delta(u, v)$ separati da una barra. Il valore $d_{uv} = \delta(u, v)$ è uguale a $\hat{\delta}(u, v) + h(v) - h(u)$.*

Calcolo dei cammini minimi tra tutte le coppie

L'algoritmo di Johnson per calcolare i cammini minimi tra tutte le coppie usa l'algoritmo di Bellman-Ford (paragrafo 25.3) e quello di Dijkstra (Paragrafo 25.2) come sottoprocedure. Esso assume che gli archi siano memorizzati in liste di adiacenza. L'algoritmo restituisce la solita matrice $D = d_{ij}$ di dimensione $|V| \times |V|$, dove $d_{ij} = \delta(i, j)$, oppure comunica che il grafo di input contiene un ciclo di peso negativo. (Per fare in modo che gli indici della matrice D abbiano senso, si assume che i vertici siano numerati da 1 a $|V|$).

```
JOHNSON(G)
1   si calcola G', dove V[G'] = V[G] ∪ {s} e
        E[G'] = E[G] ∪ {(s, v) : v ∈ V[G]}
2   if BELLMAN-FORD(G', w, s) = FALSE
3       then stampa "il grafo di input contiene un ciclo di peso negativo"
4       else  for ogni vertice v ∈ V[G']
5                 do assegna ad h(v) il valore δ(s, v)
                        calcolato dall'algoritmo di Bellman-Ford
6             for ogni arco (u, v) ∈ E[G']
7                 do ŵ(u, v) ← w(u, v) + h(u) − h(v)
8             for ogni vertice u ∈ V[G]
9                 do  esegui DIJKSTRA(G, ŵ, u) per calcolare
                          δ̂(u, v) per tutti i v ∈ V[G]
10                    for ogni vertice v ∈ V[G]
11                        do d_uv ← δ̂(u, v) + h(v) − h(u)
12  return D
```

Questo codice non fa altro che eseguire le azioni che abbiamo descritto sopra. La linea 2 esegue l'algoritmo di Bellman-Ford su G' con funzione peso w; se G' (e quindi anche G) contiene un ciclo di peso negativo, la linea 3 rileva il problema. Le linee 4-11 assumono che G' non contenga cicli di peso negativo. Le linee 4 e 5 assegnano ad $h(v)$ il peso di cammino minimo $\delta(s, v)$ calcolato dall'algoritmo di Bellman-Ford per tutti i $v \in V'$; le linee 6 e 7 calcolano i nuovi pesi $\hat{w}$. Per ogni coppia di vertici $u, v \in V$, il ciclo **for** delle linee 8-11 calcola il peso di cammino minimo $\delta(u, v)$ chiamando l'algoritmo di Dijkstra una volta per ogni vertice in V. La linea 11 memorizza nell'elemento di matrice d_{uv} il corretto peso di cammino minimo $\delta(u, v)$ calcolato usando l'equazione (26.10). Infine, la linea 12 restituisce la matrice D completata. La figura 26.6 mostra l'esecuzione dell'algoritmo di Johnson.

Si può vedere facilmente che il tempo di esecuzione dell'algoritmo di Johnson è $O(V^2 \lg V + VE)$, se la coda con priorità nell'algoritmo di Dijkstra viene realizzata con uno heap di Fibonacci. Una realizzazione più semplice con uno heap binario porta ad un tempo di esecuzione $O(VE \lg V)$ che è ancora asintoticamente minore del tempo di esecuzione dell'algoritmo di Floyd-Warshall se il grafo è sparso.

Esercizi

26.3-1 Si usi l'algoritmo di Johnson per trovare i cammini minimi tra tutte le coppie di vertici nel grafo di figura 26.2. Si mostrino i valori di h e $\hat{w}$ calcolati dall'algoritmo.

26.3-2 Per quale motivo si aggiunge il nuovo vertice s a V, ottenendo V'?

26.3-3 Si supponga che $w(u, v) \geq 0$ per tutti gli archi $(u, v) \in E$. Qual è la relazione tra le funzioni peso w e $\hat{w}$?

* 26.4 Un contesto generale in cui risolvere problemi di cammini in grafi orientati

In questo paragrafo esamineremo i "semianelli chiusi", una struttura algebrica che rappresenta un contesto generale in cui risolvere problemi di cammini in grafi orientati. Cominceremo con la definizione di semianello chiuso e discuteremo la relazione col calcolo di cammini orientati. Quindi, mostreremo alcuni esempi di semianelli chiusi ed un algoritmo "generico" per calcolare informazioni su cammini tra tutte le coppie. Sia l'algoritmo di Floyd-Warshall che quello della chiusura transitiva del paragrafo 26.2 sono istanze di questo algoritmo generico.

Definizione di semianelli chiusi

Un ***semianello chiuso*** è un sistema $(S, \oplus, \odot, \bar{0}, \bar{1})$, dove S è un insieme di elementi, $\oplus$ (l'***operatore additivo***) e $\odot$ (l'***operatore di estensione***) sono operatori binari su S, $\bar{0}$ ed $\bar{1}$ sono elementi di S, e le seguenti otto proprietà sono soddisfatte:

1. $(S, \oplus, \bar{0})$ è un ***monoide***:
   - S è ***chiuso*** rispetto a $\oplus$: per tutti gli $a, b \in S$, $a \oplus b \in S$.
   - $\oplus$ è ***associativo***: per tutti gli $a, b, c \in S$, $a \oplus (b \oplus c) = (a \oplus b) \oplus c$.
   - $\bar{0}$ è l'***identità*** per $\oplus$: per ogni $a \in S$, $a \oplus \bar{0} = \bar{0} \oplus a = a$.

   Analogamente, anche $(S, \odot, \bar{1})$ è un monoide:
2. $\bar{0}$ è un ***annullatore***: per tutti gli $a \in S$, $a \odot \bar{0} = \bar{0} \odot a = \bar{0}$.
3. $\oplus$ è ***commutativo***: per tutti gli $a, b \in S$, $a \oplus b = b \oplus a$.
4. $\oplus$ è ***idempotente***: per tutti gli $a \in S$, $a \oplus a = a$.
5. $\odot$ ***distribuisce*** rispetto a $\oplus$: per tutti gli $a, b, c \in S$, $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ e $(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$.
6. Se $a_1, a_2, a_3, \ldots$ è una sequenza numerabile di elementi di S, allora $a_1 \oplus a_2 \oplus a_3 \ldots$ è ben definito in S.
7. Associatività, commutatività ed idempotenza si applicano a sommatorie infinite. (Quindi, una qualunque sommatoria infinita può essere riscritta come una sommatoria infinita in cui ogni termine della sommatoria compare una sola volta e l'ordine di valutazione è arbitrario).
8. $\odot$ distribuisce rispetto a sommatorie infinite: $a \odot (b_1 \oplus b_2 \oplus b_3 \oplus \ldots) = (a \odot b_1) \oplus (a \odot b_2) \oplus (a \odot b_3) \oplus \ldots$, e $(b_1 \oplus b_2 \oplus b_3 \oplus \ldots) \odot a = (b_1 \odot a) \oplus (b_2 \odot a) \oplus (b_3 \odot a) \oplus \ldots$

Un calcolo di cammini in grafi orientati

Anche se le proprietà dei semianelli chiusi possono sembrare astratte, esse possono essere messe in relazione col calcolo di cammini in grafi orientati. Si supponga di avere un grafo orientato $G = (V, E)$ ed una ***funzione di etichettatura*** $\lambda : V \times V \to S$ che associa ad ogni coppia ordinata di vertici un elemento di un qualche codominio S.

L'*etichetta di un arco* $(u, v) \in E$ è denotata con $\lambda(u, v)$; poiché λ è definita su tutto il dominio $V \times V$, l'etichetta $\lambda(u, v)$ vale, di solito, $\bar{0}$ se (u, v) non è un arco di G (vedremo tra un attimo perché).

Si usa l'operatore associativo di estensione $\odot$ per estendere la nozione di etichetta ai cammini. L'*etichetta di un cammino* $p = \langle v_1, v_2, \ldots, v_k \rangle$ è data da

$$\lambda(p) = \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k) .$$

L'identità $\bar{1}$ dell'operatore $\odot$ serve come etichetta del cammino vuoto.

Come esempio di applicazione dei semianelli chiusi, considereremo i cammini minimi con pesi di archi non negativi. Il codominio S è in questo caso $\mathbb{R}^{\geq 0} \cup \{\infty\}$, dove $\mathbb{R}^{\geq 0}$ è l'insieme dei reali non negativi e $\lambda(i, j) = w_{ij}$ per tutti gli $i, j \in V$. L'operatore di estensione $\odot$ corrisponde all'operatore aritmetico + e l'etichetta del cammino $p = \langle v_1, v_2, \ldots, v_k \rangle$ è di conseguenza

$$\begin{aligned} \lambda(p) &= \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k) \\ &= w_{v_1, v_2} + w_{v_2, v_3} + \cdots + w_{v_{k-1}, v_k} \\ &= w(p) . \end{aligned}$$

Naturalmente, il ruolo di $\bar{1}$, l'identità per $\odot$, è preso da 0, l'identità per +; denoteremo il cammino vuoto con ε e la sua etichetta è $\lambda(\varepsilon) = w(\varepsilon) = 0 = \bar{1}$.

Poiché l'operatore di estensione $\odot$ è associativo, possiamo definire l'etichetta della concatenazione di due cammini in modo naturale. Dati i cammini $p_1 = \langle v_1, v_2, \ldots, v_k \rangle$ e $p_2 = \langle v_k, v_{k+1}, \ldots, v_l \rangle$, la loro *concatenazione* è

$$p_1 \circ p_2 = \langle v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, v_l \rangle ,$$

e l'etichetta della loro concatenazione è

$$\begin{aligned} \lambda(p_1 \circ p_2) &= \lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k) \odot \\ &\qquad \lambda(v_k, v_{k+1}) \odot \lambda(v_{k+1}, v_{k+2}) \odot \cdots \odot \lambda(v_{l-1}, v_l) \\ &= (\lambda(v_1, v_2) \odot \lambda(v_2, v_3) \odot \cdots \odot \lambda(v_{k-1}, v_k)) \odot \\ &\qquad (\lambda(v_k, v_{k+1}) \odot \lambda(v_{k+1}, v_{k+2}) \odot \cdots \odot \lambda(v_{l-1}, v_l)) \\ &= \lambda(p_1) \odot \lambda(p_2) . \end{aligned}$$

L'operatore additivo $\oplus$, che è sia commutativo che associativo, è usato per ***riassumere*** le etichette di cammini. In altre parole, il valore $\lambda(p_1) \oplus \lambda(p_2)$ viene visto come un 'riassunto' delle etichette dei cammini p_1 e p_2, la semantica del quale dipende dall'applicazione.

Il nostro obiettivo sarà di calcolare, per tutte le coppie di vertici $i, j \in V$, il riassunto di tutte le etichette di cammini da i a j:

$$l_{ij} = \bigoplus_{p: i \rightsquigarrow j} \lambda(p) . \tag{26.11}$$

Si richiedono la commutatività e l'associatività di $\oplus$ perché l'ordine in cui i cammini vengono riassunti deve essere ininfluente. Inoltre, poiché usiamo l'annullatore $\bar{0}$ come etichetta di ogni coppia ordinata (u, v) che non sia un arco del grafo, qualunque cammino cerchi di seguire un arco non esistente avrà $\bar{0}$ come etichetta.

Figura 26.7 *Uso della distributività di $\odot$ rispetto a $\oplus$. Per riassumere le etichette dei cammini $p_1 \circ p_2$ e $p_1 \circ p_3$, possiamo calcolare o $(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(p_3))$, oppure $\lambda(p_1) \odot (\lambda(p_2) \oplus \lambda(p_3))$.*

Per i cammini minimi useremo "min" nel ruolo dell'operatore additivo $\oplus$. L'identità per min è ∞ e, in effetti, ∞ è un annullatore per +: infatti $a + \infty = \infty + a = \infty$, per tutti gli $a \in \mathbb{R}^{\geq 0} \cup \{\infty\}$. Archi non esistenti hanno peso ∞ e se un qualunque arco di un cammino ha peso ∞ anche il cammino avrà peso ∞.

Si vuole che l'operatore additivo sia idempotente perché dall'equazione (26.11) si nota che $\oplus$ deve riassumere le etichette di un *insieme* di cammini. Se p è un cammino, allora $\{p\} \cup \{p\} = \{p\}$; se 'riassumiamo' il cammino p con se stesso, l'etichetta risultante deve essere l'etichetta di p, cioè $\lambda(p) \oplus \lambda(p) = \lambda(p)$.

Poiché consideriamo cammini che possono non essere semplici, ci può essere un numero infinito numerabile di cammini in un grafo. (Ogni cammino, sia semplice che non semplice, ha un numero finito di archi). L'operatore $\oplus$ deve quindi essere applicabile ad un numero infinito numerabile di etichette di cammini; più precisamente, se $a_1, a_2, a_3, \ldots$ è una sequenza numerabile di elementi del codominio S, allora l'etichetta $a_1 \oplus a_2 \oplus a_3 \oplus \ldots$ deve essere ben definita ed in S. Deve essere ininfluente l'ordine in cui vengono riassunte le etichette dei cammini, quindi l'associatività e la commutatività deve valere per sommatorie infinite.

Tornando all'esempio dei cammini minimi, si richiede che min sia applicabile ad una sequenza infinita di valori in $\mathbb{R}^{\geq 0} \cup \{\infty\}$. Per esempio, è ben definito il valore di $\min_{k=1}^{\infty} \{1/k\}$? Lo è se si assume che l'operatore min restituisca in effetti l'estremo inferiore dei suoi argomenti: in questo caso abbiamo $\min_{k=1}^{\infty} \{1/k\} = 0$.

Per calcolare le etichette di cammini che divergono, abbiamo bisogno della distributività dell'operatore di estensione $\odot$ rispetto all'operatore additivo $\oplus$. Come mostrato nella figura 26.7, si supponga di avere i cammini $u \xrightarrow{p_1} v$, $v \xrightarrow{p_2} x$ e $v \xrightarrow{p_3} y$. Per la distributività, possiamo riassumere le etichette dei cammini $p_1 \circ p_2$ e $p_1 \circ p_3$ calcolando o $\lambda(p_1) \odot (\lambda(p_2)) \oplus \lambda(p_1) \odot \lambda(p_3))$ oppure $\lambda(p_1) \odot (\lambda(p_2) \oplus \lambda(p_3))$.

Poiché ci può essere un numero infinito numerabile di cammini in un grafo, $\odot$ deve distribuire anche rispetto a sommatorie infinite. La figura 26.8, per esempio, contiene i cammini $u \xrightarrow{p_1} v$ e $v \xrightarrow{p_2} x$, ed il ciclo $v \xrightarrow{c} v$. Dobbiamo essere in grado di riassumere i cammini $p_1 \circ p_2, p_1 \circ c \circ p_2, p_1 \circ c \circ c \circ p_2, \ldots$: la distributività di $\odot$ rispetto a sommatorie infinite ci dà

$$\begin{aligned} &(\lambda(p_1) \odot \lambda(p_2)) \oplus (\lambda(p_1) \odot \lambda(c) \odot \lambda(p_2)) \\ &\qquad \oplus (\lambda(p_1) \odot \lambda(c) \odot \lambda(c) \odot \lambda(p_2)) \oplus \cdots \\ &= \lambda(p_1) \odot (\lambda(p_2) \oplus (\lambda(c) \odot \lambda(p_2)) \oplus (\lambda(c) \odot \lambda(c) \odot \lambda(p_2)) \oplus \cdots) \\ &= \lambda(p_1) \odot (\bar{1} \oplus \lambda(c) \oplus (\lambda(c) \odot \lambda(c)) \oplus (\lambda(c) \odot \lambda(c) \odot \lambda(c)) \oplus \cdots) \odot \lambda(p_2) . \end{aligned}$$

Figura 26.8 *Distributività di $\odot$ rispetto a sommatorie infinite numerabili di $\oplus$. A causa del ciclo c, vi è un insieme infinito numerabile di cammini dal vertice v al vertice x. Dobbiamo, infatti, essere in grado di riassumere i cammini $p_1 \circ p_2$, $p_1 \circ c \circ p_2$, $p_1 \circ c \circ c \circ p_2$, ...*

Useremo una notazione speciale per denotare l'etichetta di un ciclo che può essere percorso un numero arbitrario di volte. Si supponga di avere un ciclo c con etichetta $\lambda(c) = a$: si può percorrere c zero volte con una corrispondente etichetta $\lambda(\varepsilon) = \bar{1}$, una volta con etichetta $\lambda(c) = a$, due volte con etichetta $\lambda(c) \odot \lambda(c) = a \odot a$ e così via. L'etichetta che otteniamo riassumendo questo numero infinito di percorrimenti del ciclo c è la ***chiusura*** di a definita come

$$a^* = \bar{1} \oplus a \oplus (a \odot a) \oplus (a \odot a \odot a) \oplus (a \odot a \odot a \odot a) \oplus \cdots .$$

Quindi, nella figura 26.8 si vuole calcolare $\lambda(p_1) \odot (\lambda(c))^* \odot \lambda(p_2)$.

Tornando all'esempio dei cammini minimi, per qualunque reale non negativo $a \in \mathbf{R}^{\geq 0} \cup \{\infty\}$, abbiamo

$$\begin{aligned} a^* &= \min_{k=0}^{\infty} \{ka\} \\ &= 0 . \end{aligned}$$

L'interpretazione di questa proprietà è la seguente: poiché tutti i cicli hanno peso non negativo, nessun cammino minimo ha mai bisogno di percorrere un intero ciclo.

Esempi di semianelli chiusi

Abbiamo già visto un esempio di semianello chiuso e cioè $S_1 = (\mathbf{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$ che abbiamo usato per i cammini minimi con pesi degli archi non negativi. (Come osservato precedentemente, l'operatore min restituisce in realtà l'estremo inferiore dei suoi argomenti). Abbiamo anche mostrato che $a^* = 0$ per ogni $a \in \mathbf{R}^{\geq 0} \cup \{\infty\}$.

Tuttavia avevamo affermato che, anche se vi sono archi di peso negativo, l'algoritmo di Floyd-Warshall calcola i pesi di cammino minimo purché non vi siano cicli di peso negativo. Aggiungendo l'operatore di chiusura appropriato ed estendendo il codominio delle etichette a $\mathbf{R} \cup \{-\infty, +\infty\}$, possiamo trovare un semianello chiuso che ci permette di trattare cicli di peso negativo. Usando min per $\oplus$ e + per $\odot$, il lettore può verificare che la chiusura di $a \in \mathbf{R} \cup \{-\infty, +\infty\}$ è

$$a^* = \begin{cases} 0 & \text{se } a \geq 0 , \\ -\infty & \text{se } a < 0 . \end{cases}$$

Il secondo caso ($a < 0$) modella la situazione in cui possiamo percorrere un ciclo di peso negativo un numero infinito di volte, ottenendo un peso di $-\infty$ per ogni cammino che contenga il ciclo. Quindi, il semianello chiuso da usare per l'algoritmo di Floyd-Warshall con pesi negativi degli archi è $S_2 = (\mathbf{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$ (si veda l'Esercizio 26.4-3).

Per la chiusura transitiva, si usa il semianello chiuso $S_3 = (\{0, 1\}, \vee, \wedge, 0, 1)$, dove $\lambda(i, j) = 1$ se $(i, j) \in E$, e $\lambda(i, j) = 0$ altrimenti; in questo caso abbiamo $0^* = 1^* = 1$.

Un algoritmo per etichette di cammini orientati mediante programmazione dinamica

Sia $G = (V, E)$ un grafo orientato con funzione di etichettatura $\lambda : V \times V \to S$ ed i cui vertici sono numerati da 1 a n. Per ogni coppia di vertici $i, j \in V$, si vuole calcolare l'equazione (26.11):

$$l_{ij} = \bigoplus_{p \in \mathcal{L}_{ij}} \lambda(p) ,$$

che è il risultato di riassumere tutti i cammini da i a j usando l'operatore additivo $\oplus$. Per i cammini minimi, ad esempio, vorremmo calcolare

$$l_{ij} = \delta(i, j) = \min_{p \in \mathcal{L}_{ij}} \{w(p)\} .$$

Mediante tecniche di programmazione dinamica, è possibile costruire un algoritmo che risolve questo problema e la sua forma è molto simile a quella degli algoritmi di Floyd-Warshall e della chiusura transitiva. Sia $Q_{ij}^{(k)}$ l'insieme dei cammini dal vertice i al vertice j aventi tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k\}$. Si definisce

$$l_{ij}^{(k)} = \bigoplus_{p \in Q_{ij}^{(k)}} \lambda(p) .$$

Si noti l'analogia con la definizione di $d_{ij}^{(k)}$ nell'algoritmo di Floyd-Warshall e di $t_{ij}^{(k)}$ nell'algoritmo della chiusura transitiva. Possiamo definire $l_{ij}^{(k)}$ ricorsivamente come

$$l_{ij}^{(k)} = l_{ij}^{(k-1)} \oplus \left(l_{ik}^{(k-1)} \odot (l_{kk}^{(k-1)})^* \odot l_{kj}^{(k-1)} \right) . \tag{26.12}$$

La ricorrenza (26.12) ricorda le ricorrenze (26.5) e (26.8), ma contiene il fattore addizionale $l_{kk}^{(k-1)}$; questo fattore rappresenta il riassunto di tutti i cicli che passano per il vertice k e che hanno tutti gli altri vertici nell'insieme $\{1, 2, \ldots, k-1\}$. (Quando nell'algoritmo di Floyd-Warshall assumiamo che non vi siano cicli di peso negativo, abbiamo che $(c_{kk}^{(k-1)})^*$ è 0, che corrisponde a $\bar{1}$, il peso di un ciclo vuoto. Nell'algoritmo della chiusura transitiva, il cammino vuoto da k a k ci dà $(t_{kk}^{(k-1)})^* = 1 = \bar{1}$. Quindi, per entrambi questi algoritmi possiamo ignorare il fattore $(l_{kk}^{(k-1)})^*$ perché esso non è altro che l'identità per $\odot$.) La base della definizione ricorsiva è

$$l_{ij}^{(0)} = \begin{cases} \lambda(i, j) & \text{se } i \neq j , \\ \bar{1} \oplus \lambda(i, j) & \text{se } i = j , \end{cases}$$

che possiamo interpretare nel modo seguente. L'etichetta del cammino costituito da un solo arco (i, j) è semplicemente $\lambda(i, j)$ (che è uguale a $\bar{0}$ se (i, j) non è un arco in E). Se inoltre $i = j$, allora $\bar{1}$ è l'etichetta del cammino vuoto da i a i.

L'algoritmo di programmazione dinamica calcola i valori $l_{ij}^{(k)}$ per valori crescenti di k e restituisce la matrice $L^{(n)} = \left(l_{ij}^{(n)}\right)$.

```
COMPUTE-SUMMARIES(λ, V)
n ← |V|
for i ← 1 to n
    do for j ← 1 to n
        do if i = j
            then  l_ij^(0) ← 1̄ ⊕ λ(i, j)
            else  l_ij^(0) ← λ(i, j)
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            do l_ij^(k) ← l_ij^(k-1) ⊕ (l_ik^(k-1) ⊙ (l_kk^(k-1))* ⊙ l_kj^(k-1))
return L^(n)
```

Il tempo di esecuzione di questo algoritmo dipende dal tempo richiesto per calcolare $\odot$, $\oplus$ e *. Se denotiamo con $T_{\odot}$, $T_{\oplus}$, e T_* i rispettivi tempi, allora il tempo di esecuzione di COMPUTE-SUMMARIES è $\Theta(n^3(T_{\odot} + T_{\oplus} + T_*))$ che è $\Theta(n^3)$ se ognuna delle tre operazioni richiede tempo $O(1)$.

Esercizi

26.4-1 Si verifichi che $S_1 = (\mathbf{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$ e $S_3 = (\{0, 1\}, \vee, \wedge, 0, 1)$ sono semianelli chiusi.

26.4-2 Si verifichi che $S_2 = (\mathbf{R} \cup \{-\infty, +\infty\}, \min, +, +\infty, 0)$ è un semianello chiuso. Qual è il valore di $a + (-\infty)$ per $a \in \mathbf{R}$? Cosa succede con $(-\infty) + (+\infty)$?

26.4-3 Si riscriva la procedura COMPUTE-SUMMARIES usando il semianello chiuso S_2 in modo che realizzi l'algoritmo di Floyd-Warshall. Quale dovrebbe essere il valore di $-\infty + \infty$?

26.4-4 Il sistema $S_4 = (\mathbf{R}, +, \cdot, 0, 1)$ è un semianello chiuso?

26.4-5 Si può usare un semianello chiuso arbitrario per l'algoritmo di Dijkstra? E per l'algoritmo di Bellman-Ford? E cosa succede per la procedura FASTER-ALL-PAIRS-SHORTEST-PATHS?

26.4-6 Una ditta di trasporti vuole inviare un camion da Castroville a Boston con il massimo carico possibile di carciofi, ma ogni strada negli Stati Uniti ha un limite massimo di peso per i camion che la percorrono. Si modelli questo problema con un grafo orientato $G = (V, E)$ e con un appropriato semianello chiuso e si fornisca un algoritmo efficiente per risolverlo.

Problemi

26-1 Chiusura transitiva di un grafo dinamico

Si supponga di voler mantenere la chiusura transitiva di un grafo orientato $G = (V, E)$ quando si inseriscono dei nuovi archi in E; in altre parole, dopo che un arco è stato inserito, vogliamo aggiornare la chiusura transitiva degli archi inseriti sino a quel momento. Si assuma che all'inizio il grafo G non abbia alcun arco e che la chiusura transitiva venga rappresentata come una matrice booleana.

a. Si mostri come si può aggiornare la chiusura transitiva $G^* = (V, E^*)$ di un grafo $G = (V, E)$ in tempo $O(V^2)$ quando viene aggiunto un nuovo arco a G.

b. Si dia un esempio di un grafo G e di un arco e tali che sia richiesto tempo $\Omega(V^2)$ per aggiornare la chiusura transitiva dopo l'inserimento di e in G.

c. Si descriva un algoritmo efficiente per aggiornare la chiusura transitiva quando gli archi vengono inseriti nel grafo. Per una qualunque sequenza di n inserzioni, l'algoritmo proposto dovrebbe richiedere un tempo complessivo di $\sum_{i=1}^{n} t_i = O(V^3)$, dove t_i è il tempo necessario per aggiornare la chiusura transitiva quando viene inserito l'i-esimo arco. Si dimostri che l'algoritmo proposto raggiunge questo limite di tempo.

26-2 Cammini minimi in grafi ε-densi

Un grafo $G = (V, E)$ è ***ε-denso*** se $|E| = \Theta(V^{1+\varepsilon})$ per una qualche costante ε nell'intervallo $0 < \varepsilon \leq 1$. Usando uno heap d-ario (si veda il Problema 7-2) in algoritmi di cammini minimi per grafi ε-densi, si può raggiungere l'efficienza degli algoritmi basati su heap di Fibonacci senza usare una struttura di dati tanto complessa.

a. Qual è il tempo di esecuzione asintotico per INSERT, EXTRACT-MIN e DECREASE-KEY in funzione di d e del numero n di elementi in uno heap d-ario? Come diventano questi tempi di esecuzione se si sceglie $d = \Theta(n^{\alpha})$, per una qualche costante $0 < \alpha < 1$? Si confrontino questi tempi di esecuzione con i costi ammortizzati di queste operazioni per uno heap di Fibonacci.

b. Si mostri come si possono calcolare in tempo $O(E)$ i cammini minimi da una sorgente singola per un grafo orientato ed ε-denso $G = (V, E)$ senza archi di peso negativo. (*Suggerimento*: si scelga d in funzione di ε).

c. Si mostri come risolvere in tempo $O(VE)$ il problema di cammini minimi tra tutte le coppie per un grafo orientato ed ε-denso $G = (V, E)$ senza archi di peso negativo.

d. Si mostri come risolvere il problema in tempo $O(VE)$ di cammini minimi tra tutte le coppie per un grafo orientato ed ε-denso $G = (V, E)$ che può avere archi di peso negativo ma che non ha cicli di peso negativo.

26-3 Albero di copertura minimo come un semianello chiuso

Sia $G = (V, E)$ un grafo connesso e non orientato con funzione peso $w : E \to \mathbf{R}$. Sia $V = \{1, 2, \ldots, n\}$ l'insieme dei vertici, dove $n = |V|$, e si assuma che tutti i pesi $w(i, j)$ degli archi

siano distinti. Sia T l'unico albero di copertura minimo di G (si veda l'Esercizio 24.1-6). In questo problema determineremo T usando un semianello chiuso, come suggerito da B. M. Maggs e S. A. Plotkin. Dapprima determiniamo, per ogni coppia di vertici $i, j \in V$, il peso *minmax*

$$m_{ij} = \min_{p \in \mathcal{P}_{ij}} \max_{\text{archi } e \text{ in } p} w(e) .$$

a. Giustificare brevemente l'asserzione che $S = (\mathbf{R} \cup \{-\infty, +\infty\}, \min, \max, \infty, -\infty)$ è un semianello chiuso.

Poiché S è un semianello chiuso, possiamo usare la procedura COMPUTE-SUMMARIES per determinare i pesi minmax m_{ij} nel grafo G. Sia $m_{ij}^{(k)}$ il peso minmax su tutti i cammini dal vertice i al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, \ldots, k\}$.

b. Si dia una ricorrenza per $m_{ij}^{(k)}$, dove $k \geq 0$.

c. Sia $T_m = \{(i, j) \in E : w(i, j) = m^{ij}\}$. Si dimostri che gli archi in T_m formano un albero di copertura di G.

d. Si mostri che $T_m = T$. (*Suggerimento*: si consideri l'effetto di aggiungere un arco (i, j) a T e di rimuovere un arco su un altro cammino da i a j. Si consideri anche l'effetto di rimuovere un arco (i, j) da T e di sostituirlo con un altro arco).

Note al capitolo

Lawler [132] presenta una buona discussione del problema di cammini minimi tra tutte le coppie anche se non analizza soluzioni per grafi sparsi; egli attribuisce l'algoritmo basato sulla moltiplicazione di matrici al folklore. L'algoritmo di Floyd-Warshall è dovuto a Floyd [68] ed è basato su un teorema di Warshall [198] che descrive il calcolo della chiusura transitiva di matrici booleane. La struttura algebrica di semianello chiuso compare in Aho, Hopcroft e Ullman [4]. L'algoritmo di Johnson è tratto da [114].

Flusso massimo

27

Proprio come possiamo modellare una carta stradale con un grafo orientato allo scopo di trovare il cammino minimo da un punto ad un altro, possiamo anche interpretare un grafo orientato come una "rete di flusso" ed usarlo per rispondere a questioni riguardanti flussi di materiale. Si immagini che una certa quantità di materiale scorra attraverso un sistema a partire da una sorgente, dove il materiale viene prodotto, fino ad un pozzo, dove esso viene consumato. La sorgente produce materiale ad un ritmo costante ed il pozzo lo consuma alla stessa velocità. Il "flusso" del materiale in un qualunque punto nel sistema è intuitivamente la velocità con la quale il materiale si muove in quel punto. Le reti di flusso possono essere usate per modellare lo scorrere di liquidi attraverso tubazioni, di componenti attraverso catene di montaggio, di corrente attraverso reti elettriche, di informazioni attraverso reti di comunicazione, e così via.

Ogni arco orientato di una rete di flusso può essere considerato come un condotto per il materiale; ogni condotto ha una capacità prefissata, che stabilisce la quantità massima di materiale che può passare attraverso di esso in un'opportuna unità di tempo: per esempio, 2000 litri di liquido all'ora per una tubazione o 20 Ampere di corrente elettrica attraverso un filo elettrico. I vertici del grafo rappresentano giunti tra i condotti e, ad eccezione della sorgente e del pozzo, il materiale scorre attraverso i vertici senza accumularvisi; in altri termini, la velocità con cui il materiale entra in un vertice deve essere uguale alla velocità con cui ne esce. Chiameremo questa proprietà "conservazione del flusso": se il materiale è la corrente elettrica, questa proprietà è esattamente la legge di Kirchhoff.

Il problema di flusso massimo è il problema più semplice tra quelli che riguardano reti di flusso. Esso chiede semplicemente quale sia la velocità massima con cui il materiale può essere inviato dalla sorgente al pozzo senza violare i vincoli di capacità. Come vedremo in questo capitolo, il problema può essere risolto con algoritmi efficienti ed inoltre le tecniche di base usate da questi algoritmi possono essere adattate per risolvere altri problemi su reti di flusso.

Questo capitolo presenta due metodi generali per risolvere il problema di flusso massimo. Il paragrafo 27.1 formalizza la nozione di rete di flusso e di flusso, definendo formalmente il problema di flusso massimo. Il paragrafo 27.2 descrive il metodo classico di Ford e Fulkerson per trovare i flussi massimi. Un'applicazione di questo metodo, consistente nel trovare un abbinamento massimo in un grafo bipartito non orientato, viene presentata nel paragrafo 27.3. Il paragrafo 27.4 presenta il metodo dei preflussi sul quale si basano molti dei più veloci algoritmi per problemi di reti di flusso. Il paragrafo 27.5 tratta l'algoritmo "lift-to-front", una particolare realizzazione del metodo dei preflussi che richiede tempo $O(V^3)$.

Figura 27.1 (a) *Una rete di flusso $G = (V, E)$ per il problema di trasporto della Lucky Puck Company. La fabbrica a Vancouver è la sorgente s e il deposito a Winnipeg è il pozzo t. I dischi da hockey vengono spediti attraverso città intermedie, ma solo $c(u, v)$ casse al giorno possono viaggiare dalla città u alla città v. Ogni arco è etichettato con la sua capacità.* (b) *Un flusso f in G con valore $|f| = 19$: sono mostrati solo i flussi netti positivi. Se $f(u, v) > 0$, allora l'arco (u, v) è etichettato con $f(u, v) / c(u, v)$. (La barra è usata solo per dividere flusso e capacità: non indica una divisione). Se $f(u, v) \leq 0$, allora l'arco (u, v) è etichettato solo con la capacità.*

Anche se questo algoritmo non è il più veloce tra quelli conosciuti, esso illustra alcune delle tecniche usate negli algoritmi asintoticamente più veloci ed è ragionevolmente efficiente nella pratica.

27.1 Reti di flusso

In questo paragrafo diamo una definizione di reti di flusso usando la teoria dei grafi, discutiamo le loro proprietà e definiamo rigorosamente il problema di flusso massimo. Inoltre, introduciamo alcune utili notazioni.

Reti di flusso e flussi

Una ***rete di flusso*** $G = (V, E)$ è un grafo orientato in cui ogni arco $(u, v) \in E$ ha una ***capacità*** non negativa $c(u, v) \geq 0$. Se $(u, v) \notin E$, assumiamo che $c(u, v) = 0$. In una rete di flusso vi sono due vertici speciali: una ***sorgente*** s ed un ***pozzo*** t. Per convenienza, si assume che ogni vertice giaccia su un qualche cammino dalla sorgente al pozzo cioè, per ogni vertice $v \in V$, vi è un cammino $s \rightsquigarrow v \rightsquigarrow t$. Il grafo è quindi connesso e $|E| \geq |V| - 1$. La figura 27.1 mostra un esempio di rete di flusso.

Siamo ora pronti a definire la nozione di flusso più formalmente. Sia $G = (V, E)$ una rete di flusso (con una funzione capacità implicita c). Sia s la sorgente della rete e sia t il pozzo. Un ***flusso*** in G è una funzione a valori reali $f: V \times V \rightarrow \mathbf{R}$ che soddisfa le seguenti tre proprietà:

Vincolo di capacità: Per tutti gli $u, v \in V$, si richiede che $f(u, v) \leq c(u, v)$.

Antisimmetria: Per tutti gli $u, v \in V$, si richiede che $f(u, v) = -f(v, u)$.

Conservazione del flusso: Per tutti gli $u \in V - \{s, t\}$, si richiede che

$$\sum_{v \in V} f(u, v) = 0 .$$

La quantità $f(u, v)$, che può essere positiva o negativa, è chiamata ***flusso netto*** dal vertice u al vertice v. Il ***valore*** di un flusso f è definito come

$$|f| = \sum_{v \in V} f(s, v) , \tag{27.1}$$

cioè il flusso netto totale che esce dalla sorgente. (Qui la notazione $|\cdot|$ denota il valore del flusso, non il valore assoluto o la cardinalità.) Nel ***problema di flusso massimo*** viene data una rete di flusso G con sorgente s e pozzo t e si vuole trovare un flusso di valore massimo da s a t.

Prima di vedere un esempio di un problema di flusso massimo, esaminiamo brevemente le tre proprietà. Il vincolo di capacità dice semplicemente che il flusso netto da un vertice ad un altro non può eccedere la corrispondente capacità. L'antisimmetria dice che il flusso netto da un vertice u ad un vertice v è l'opposto del flusso netto nella direzione inversa. Quindi, il flusso netto da un vertice a se stesso deve essere 0 perché, per ogni $u \in V$, abbiamo $f(u, u) = -f(u, u)$, il che implica $f(u, u) = 0$. La proprietà di conservazione del flusso dice che il flusso netto totale che esce da un vertice diverso dalla sorgente o dal pozzo è 0. Per l'antisimmetria, possiamo riscrivere la proprietà di conservazione del flusso come

$$\sum_{u \in V} f(u, v) = 0$$

per tutti i $v \in V - \{s, t\}$; quindi, il flusso netto totale entrante in un vertice è anch'esso 0.

Si osservi anche che non ci può essere flusso netto da u a v se non vi è un arco tra di essi. Se né $(u, v) \in E$ né $(v, u) \in E$, allora $c(u, v) = c(v, u) = 0$ e quindi per il vincolo di capacità $f(u, v) < 0$ e $f(v, u) < 0$. Ma poiché $f(u, v) = -f(v, u)$ per antisimmetria, abbiamo che $f(u, v) = f(u, v) = 0$. Quindi un flusso netto diverso da zero dal vertice u al vertice v implica che $(u, v) \in E$ o che $(v, u) \in E$ (oppure entrambi).

L'ultima osservazione sulle proprietà di un flusso riguarda flussi positivi. Il ***flusso netto positivo*** entrante in un vertice v è definito come

$$\sum_{\substack{u \in V \\ f(u,v) > 0}} f(u, v) . \tag{27.2}$$

Il flusso netto positivo uscente da un vertice è definito in modo simmetrico. Una interpretazione della proprietà di conservazione del flusso è che il flusso netto positivo entrante in un vertice diverso dalla sorgente o dal pozzo deve essere uguale al flusso netto positivo uscente dal vertice stesso.

Un esempio di rete di flusso

Si può usare una rete di flusso per modellare il problema di trasporto mostrato nella figura 27.1. La Lucky Puck Company ha una fabbrica (la sorgente s) a Vancouver che produce dischi di gomma per hockey su ghiaccio ed ha un magazzino (il pozzo t) a Winnipeg in cui depositarli. Lucky Puck affitta dello spazio su camion di un'altra ditta per spedire i dischi dalla fabbrica al magazzino. Poiché i camion seguono specifici percorsi tra le città ed hanno una capacità limitata, Lucky Puck può spedire al massimo $c(u, v)$ casse al giorno tra ogni

Figura 27.2 *Cancellazione.* (a) *Due vertici v_1 e v_2, con $c(v_1, v_2) = 10$ e $c(v_2, v_1) = 4$* (b) *In questo modo viene indicato il flusso netto corrispondente all'invio di 8 casse al giorno da v_1 a v_2.* (c) *Viene effettuata una spedizione addizionale di 3 casse al giorno da v_2 a v_1.* (d) *Cancellando il flusso che va in direzioni opposte, possiamo rappresentare la situazione in (c) con un flusso netto positivo in un'unica direzione.* (e) *Altre 7 casse al giorno vengono spedite da v_2 a v_1.*

coppia di città u e v della figura 27.1 (a). Lucky Puck non ha nessun controllo sui percorsi e sulle capacità e quindi non può alterare la rete di flusso mostrata nella figura 27.1 (a). Il suo scopo è di determinare il massimo numero di casse che possono essere spedite ogni giorno, e quindi di produrne tale quantità, visto che non c'è motivo di produrre più dischi di quanti ne possano essere spediti al magazzino.

La velocità con cui i dischi vengono spediti lungo ogni percorso è un flusso. I dischi escono dalla fabbrica alla velocità di p casse al giorno ed esattamente p casse devono arrivare ogni giorno al magazzino. Lucky Puck non si preoccupa di quanto tempo ci metta un determinato disco per andare dalla fabbrica al magazzino; si preoccupa solo del fatto che ogni giorno p casse escano dalla fabbrica e p casse arrivino al magazzino. I vincoli di capacità sono dati dalla restrizione che il flusso $f(u, v)$ dalla città u alla città v sia al massimo di $c(u, v)$ casse al giorno. A regime, la velocità con cui i dischi entrano in una città intermedia della rete di spedizione deve essere uguale alla velocità con cui ne escono, altrimenti vi si accumulerebbero: la legge di conservazione del flusso viene quindi rispettata. Di conseguenza un flusso massimo nella rete determina il numero massimo p di casse che possono essere spedite ogni giorno.

La figura 27.1 (b) mostra un possibile flusso nella rete, rappresentato in modo che corrisponda direttamente alle spedizioni. Per ogni coppia di vertici u e v nella rete, il flusso netto $f(u, v)$ corrisponde alla spedizione di $f(u, v)$ casse al giorno da u a v: se $f(u, v)$ è 0 o negativo, allora non vi è alcuna spedizione da u a v. Quindi, nella figura 27.1 (b) sono mostrati solo gli archi con flusso netto positivo seguiti da un segno di divisione e dalla capacità dell'arco.

Si può capire un po' meglio la relazione tra flusso netto e spedizioni concentrandosi sulle spedizioni tra due vertici. La figura 27.2(a) mostra il sottografo indotto dai vertici v_1 e v_2 nella rete di flusso della figura 27.1. Se Lucky Puck spedisce 8 casse al giorno da v_1 a v_2, allora il risultato è mostrato nella figura 27.2(b): il flusso netto da v_1 e v_2 è di 8 casse al giorno. Per l'antisimmetria, possiamo dire anche che il flusso netto da v_2 e v_1 è di -8 casse al giorno, anche se non spediamo nessun disco da v_2 e v_1. In generale, il flusso netto da v_1 a v_2 è il numero di casse spedite ogni giorno da v_1 a v_2 meno il numero di casse spedite al giorno da v_2 a v_1. La nostra convenzione per rappresentare i flussi netti è di mostrare solo flussi netti positivi perché essi indicano le reali spedizioni; quindi, solo 8 compare nella figura e non il corrispondente -8.

Aggiungiamo ora un'altra spedizione, questa volta di 3 casse al giorno, da v_2 a v_1: una naturale rappresentazione del corrispondente risultato è mostrata nella figura 27.2(c).

Abbiamo ora una situazione in cui vi sono spedizioni in entrambe le direzioni tra v_1 e v_2: si spediscono 8 casse al giorno da v_1 a v_2 e 3 casse al giorno da v_2 a v_1. Quali sono i flussi netti tra i due vertici? Il flusso netto da v_1 a v_2 è di $8 - 3 = 5$ casse al giorno, mentre quello da v_1 a v_2 è di $3 - 8 = -5$ casse al giorno.

La situazione risultante è equivalente a quella mostrata nella figura 27.2(d), in cui vengono spedite 5 casse al giorno da v_1 a v_2 e non vengono fatte spedizioni da v_2 a v_1. In effetti, le 3 casse spedite ogni giorno da v_2 a v_1 sono ***cancellate*** da 3 delle 8 casse spedite ogni giorno da v_1 a v_2. Quindi, in entrambe le situazioni, il flusso netto da v_1 a v_2 è di 5 casse al giorno, ma in (d) le reali spedizioni vengono effettuate in una sola direzione.

In generale, la cancellazione permette di rappresentare le spedizioni tra due città con un flusso netto positivo lungo al massimo uno dei due archi tra i corrispondenti vertici: se vi è un flusso nullo o negativo da un vertice ad un altro, non deve essere effettuata alcuna spedizione in quella direzione. Quindi ogni situazione in cui dei dischi vengono spediti in entrambe le direzioni tra due città può essere trasformata, usando la cancellazione, in una situazione equivalente in cui i dischi vengono spediti in una sola direzione: quella con flusso netto positivo. I vincoli di capacità non vengono violati da questa trasformazione, poiché si riducono le spedizioni in entrambe le direzioni; anche i vincoli di conservazione del flusso continuano ad essere rispettati perché il flusso netto tra ogni coppia di vertici rimane lo stesso.

Continuando con il nostro esempio, determiniamo l'effetto di spedire altre 7 casse al giorno da v_2 a v_1: la figura 27.2(e) mostra il corrispondente risultato usando la convenzione di mostrare solo i flussi netti positivi. Il flusso netto da v_1 a v_2 diventa $5 - 7 = -2$ ed il flusso netto da v_2 a v_1 diventa $7 - 5 = 2$. Poiché il flusso netto da v_2 a v_1 è positivo, esso rappresenta una spedizione di 2 casse al giorno da v_2 a v_1; invece, poiché il flusso netto da v_1 a v_2 è di -2 casse al giorno ed è negativo, non vengono spediti dischi in questa direzione. Alternativamente, possiamo pensare che 5 delle 7 casse addizionali spedite ogni giorno da v_2 a v_1 servano per cancellare la spedizione di 5 casse al giorno da v_1 a v_2 lasciando, quindi, una spedizione reale di sole 2 casse al giorno da v_2 a v_1.

Reti con sorgenti e pozzi multipli

Un problema di flusso massimo può avere diverse sorgenti e diversi pozzi invece di uno solo. La Lucky Puck Company ad esempio, può avere un insieme di m fabbriche $\{s_1, s_2, \ldots, s_m\}$ ed un insieme di n magazzini $\{t_1, t_2, \ldots, t_n\}$, come mostrato nella figura 27.3(a). Fortunatamente, questo problema non è più difficile di quello di flusso massimo ordinario.

Si può ridurre il problema di determinare un flusso massimo in una rete con sorgenti multiple e pozzi multipli ad un problema di flusso massimo ordinario. La figura 27.3(b) mostra come si può trasformare la rete (a) in una rete di flusso ordinaria con una singola sorgente ed un singolo pozzo. Si aggiunge una ***supersorgente*** s ed un arco orientato (s, s_j) con capacità $c(s, s_j) = \infty$ per ogni $j = 1, 2, \ldots, m$. Inoltre, si crea un ***superpozzo*** t e si aggiunge un arco orientato (t_j, t) con capacità $c(t_j, t) = \infty$ per ogni $j = 1, 2, \ldots, n$. Intuitivamente, ogni flusso nella rete (a) corrisponde ad un flusso nella rete (b) e viceversa. La sorgente singola s fornisce semplicemente il flusso di cui hanno bisogno le sorgenti multiple s_j ed in modo analogo il pozzo singolo t consuma tutto il flusso che dovrebbero consumare i pozzi multipli t_j. L'Esercizio 27.1-3 chiederà al lettore di dimostrare formalmente che questi due problemi sono equivalenti.

Figura 27.3 *Trasformazione di un problema di flusso massimo con sorgenti e pozzi multipli in un problema con sorgente e pozzo singoli.* (a) *Una rete di flusso con cinque sorgenti* $S = \{s_1, s_2, s_3, s_4, s_5\}$ *e tre pozzi* $T = \{t_1, t_2, t_3\}$. (b) *Una rete di flusso equivalente con una sorgente ed un pozzo. Si aggiunge una supersorgente* s' *con un arco di capacità infinita* s' *ad ognuna delle sorgenti originali. Si aggiunge, inoltre, un superpozzo* t' *con un arco di capacità infinita da ognuno dei pozzi originali a* t'.

Lavorando con i flussi

Nel seguito avremo a che fare con molte funzioni (come f) che hanno come argomenti due vertici di una rete di flusso. In questo capitolo utilizzeremo una ***notazione di sommatoria implicita*** nella quale uno dei due argomenti, o anche entrambi, possono essere un *insieme* di vertici, intendendo che il valore denotato sia la somma di tutti i possibili modi di rimpiazzare gli argomenti con i loro elementi. Ad esempio, se X e Y sono insiemi di vertici, allora

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y) .$$

Come altro esempio, il vincolo di conservazione del flusso può essere espresso come la condizione che $f(u, V) = 0$ per tutti gli $u \in V - \{s, t\}$. Inoltre, per semplicità, di solito ometteremo le parentesi graffe che delimitano un insieme se queste occorrono all'interno della notazione di sommatoria implicita. Ad esempio, nell'equazione $f(s, V - s) = f(s, V)$, il termine $V - s$ denota l'insieme $V - \{s\}$.

La notazione insiemistica implicita spesso semplifica notevolmente le equazioni che coinvolgono flussi. Il lemma seguente, la cui dimostrazione è lasciata al lettore come Esercizio 27.1-4, comprende molte delle identità riguardanti flussi e notazione insiemistica implicita che si incontrano più frequentemente.

Lemma 27.1

Sia $G = (V, E)$ un rete di flusso e sia f un flusso in G. Allora, per $X \subseteq V$,

$$f(X, X) = 0 .$$

Per $X, Y \subseteq V$,

$$f(X, Y) = -f(Y, X) .$$

Per $X, Y, Z \subseteq V$ con $X \cap Y = \emptyset$,

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$

e

$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$$

Come esempio di utilizzo della notazione di sommatoria implicita, possiamo dimostrare che il valore di un flusso è il flusso netto totale entrante nel pozzo, cioè

$$|f| = f(V, t) \tag{27.3}$$

Questo fatto è intuitivamente evidente, visto che tutti i vertici diversi dalla sorgente e dal pozzo hanno un flusso netto di 0 per la conservazione del flusso e quindi il pozzo è l'unico altro vertice che può avere un flusso netto non nullo per compensare il flusso netto non nullo della sorgente. La dimostrazione formale è la seguente:

$$\begin{aligned}
|f| &= f(s, V) && \text{(per definizione)} \\
&= f(V, V) - f(V - s, V) && \text{(per il lemma 27.1)} \\
&= f(V, V - s) && \text{(per il lemma 27.1)} \\
&= f(V, t) + f(V, V - s - t) && \text{(per il lemma 27.1)} \\
&= f(V, t) && \text{(per la conservazione del flusso).}
\end{aligned}$$

Più avanti in questo capitolo presenteremo una generalizzazione di questo risultato (lemma 27.5).

Esercizi

27.1-1 Dati due vertici u e v in una rete di flusso, dove $c(u, v) = 5$ e $c(v, u) = 8$, si supponga che 3 unità di flusso vengano inviate da u a v e 4 unità vengano inviate da v a u. Qual è il flusso netto da u a v? Si disegni la situazione nello stile della figura 27.2.

27.1-2 Si verifichi ognuna delle tre proprietà per il flusso f mostrato nella figura 27.1(b).

27.1-3 Si estendano le proprietà e le definizioni di flusso al problema con sorgenti e pozzi multipli. Si mostri che qualunque flusso in una rete di flusso con sorgenti e pozzi multipli corrisponde ad un flusso di valore identico nella rete con sorgente e pozzo singoli ottenuta aggiungendo una supersorgente ed un superpozzo e viceversa.

27.1-4 Si dimostri il lemma 27.1.

27.1-5 Per la rete di flusso $G = (V, E)$ ed il flusso f mostrati nella figura 27.1(b), si trovi una coppia di sottoinsiemi $X, Y \subseteq V$ per i quali $f(X, Y) = -f(V - X, Y)$. Si trovi poi una coppia di sottoinsiemi $X, Y \subseteq V$ per i quali $f(X, Y) \neq -f(V - X, Y)$.

27.1-6 Data una rete di flusso $G = (V, E)$, siano f_1 e f_2 due funzioni da $V \times V$ ad $\mathbf{R}$. La ***somma di flussi*** $f_1 + f_2$ è la funzione da $V \times V$ ad $\mathbf{R}$ definita come

$$(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v) \tag{27.4}$$

per tutti gli $u, v \in V$. Se f_1 e f_2 sono flussi in G, quali delle tre proprietà di flusso deve soddisfare la somma di flussi $f_1 + f_2$ e quali invece può violare?

27.1-7 Sia f un flusso in una rete e sia α un numero reale. Il ***prodotto scalare di flussi*** αf è una funzione da $V \times V$ ad $\mathbf{R}$ definita come

$$(\alpha f)(u, v) = \alpha \cdot f(u, v) .$$

Si dimostri che i flussi in una rete formano un insieme convesso mostrando che se f_1 e f_2 sono flussi, allora lo è anche $\alpha f_1 + (1 - \alpha) f_2$, per ogni α nell'intervallo $0 \leq \alpha \leq 1$.

27.1-8 Si enunci il problema di flusso massimo come un problema di programmazione lineare.

27.1-9 Il modello di rete di flusso introdotto in questo paragrafo permette il flusso di una sola merce; una ***rete di flusso per merci multiple*** consente invece il flusso di p merci distinte tra un insieme di p ***vertici sorgente*** $S = \{s_1, s_2, \ldots, s_p\}$ ed un insieme di p ***vertici pozzo*** $T = \{t_1, t_2, \ldots, t_p\}$. Il flusso netto dalla merce i-esima da u a v è denotato $f_i(u, v)$. Per la merce i-esima, l'unica sorgente è s_i e l'unico pozzo è t_i. Vi è una legge di conservazione del flusso indipendente per ogni merce: il flusso netto di ogni merce uscente da un qualunque vertice è nullo, tranne nel caso in cui il vertice è la sorgente o il pozzo per quella merce. La somma dei flussi netti di tutte le merci da u a v non deve superare $c(u, v)$, in questo modo i flussi di merci diverse interagiscono. Il valore del flusso di ogni merce è il flusso netto che esce dalla sorgente per quella merce; il ***valore di flusso totale*** è la somma dei valori dei flussi di tutte le p merci. Si dimostri che vi è un algoritmo polinomiale che risolve il problema di trovare il valore massimo di flusso totale di una rete di flusso per merci multiple, formulando il programma come un programma lineare.

27.2 Il metodo di Ford-Fulkerson

Questo paragrafo presenta il metodo di Ford-Fulkerson per risolvere il problema di flusso massimo. Lo chiamiamo un "metodo" invece che un "algoritmo" poiché esso comprende diverse realizzazioni con differenti tempi di esecuzione. Il metodo di Ford-Fulkerson dipende da tre importanti idee che trascendono il metodo stesso e che sono rilevanti per molti algoritmi e problemi di flusso: reti residue, cammini aumentanti e tagli. Queste idee sono fondamentali

per l'importante teorema "flusso massimo taglio minimo" (Teorema 27.7) che caratterizza il valore di un flusso massimo in termini dei tagli della rete di flusso. Alla fine di questo paragrafo presenteremo una particolare realizzazione del metodo di Ford-Fulkerson ed analizzeremo il suo tempo di esecuzione.

Il metodo di Ford-Fulkerson è iterativo. Si parte con $f(u, v) = 0$ per tutti gli $u, v \in V$, ottenendo un flusso iniziale di valore 0. Ad ogni iterazione si incrementa il valore del flusso cercando un "cammino aumentante", che possiamo pensare semplicemente come un cammino dalla sorgente s al pozzo t lungo il quale possiamo inviare un flusso maggiore, e quindi aumentando il flusso lungo questo cammino. Questo processo viene ripetuto finché non si può più trovare alcun cammino aumentante. Il teorema flusso massimo taglio minimo mostrerà che, al termine, questo processo produce un flusso massimo.

```
FORD-FULKERSON-METHOD(G, s, t)
inizializza il flusso f a 0
while esiste un cammino aumentante p
      do aumenta il flusso f lungo p
return f
```

Reti residue

Intuitivamente, dati una rete di flusso ed un flusso, la rete residua consiste di tutti gli archi che possono ammettere un flusso netto maggiore. Più formalmente, si supponga di avere una rete di flusso $G = (V, E)$ con sorgente s e pozzo t. Sia f un flusso in G e si consideri una coppia di vertici $u, v \in V$. La quantità di flusso netto *addizionale* che possiamo inviare da u a v prima di superare la capacità $c(u, v)$ è la ***capacità residua*** di (u, v) ed è data da

$$c_f(u, v) = c(u, v) - f(u, v) . \tag{27.5}$$

Ad esempio, se $c(u, v) = 16$ e $f(u, v) = 11$, allora possiamo inviare $c_f(u, v) = 5$ unità di flusso in più prima di superare il vincolo di capacità sull'arco (u, v). Quando il flusso netto $f(u, v)$ è negativo, la capacità residua $c_f(u, v)$ è maggiore della capacità $c(u, v)$: ad esempio, se $c(u, v) = 16$ e $f(u, v) = 4$, allora la capacità residua $c_f(u, v)$ è 20. Questo fatto si può interpretare nel modo seguente: vi è un flusso netto di 4 unità da v a u che può essere cancellato inviando un flusso netto di 4 unità da u a v; poi, si possono inviare altre 16 unità da u a v prima di violare il vincolo di capacità sull'arco (u, v). Quindi, partendo con un flusso netto di -4, si possono inviare 20 unità addizionali di flusso prima di superare la capacità consentita.

Data una rete di flusso $G = (V, E)$ ed un flusso f, la ***rete residua*** di G indotta da f è $G_f = (V, E_f)$, dove

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} .$$

Quindi, come anticipato sopra, ogni arco della rete residua, chiamato ***arco residuo***, può ammettere un flusso netto strettamente positivo. La figura 27.4(a) mostra di nuovo la rete G ed il flusso f della figura 27.1 (b). La figura 27.4(b) mostra la corrispondente rete residua G_f.

Si noti che (u, v) può essere un arco residuo in E_f anche se esso non era un arco in E: in altre parole, può succedere che $E_f \not\subseteq E$. La rete residua nella figura 27.4(b) contiene molti di questi archi che non compaiono nella rete di flusso originale, come (v_1, s) e (v_2, v_1). Un tale arco

Figura 27.4 (a) *La rete di flusso G ed il flusso f di figura 27.1(b).* (b) *La rete residua G_f con il cammino aumentante p in grigio; la sua capacità residua è $c_f(p) = c(v_2, v_3) = 4$.* (c) *Il flusso in G ottenuto aumentando il flusso della capacità residua 4 lungo il cammino aumentante p.* (d) *La rete residua indotta dal flusso in (c).*

(u, v) compare in G_f solo se $(v, u) \in E$ e vi è un flusso netto positivo da v a u. Poiché il flusso netto $f(u, v)$ da u a v è negativo, $c_f(u, v) = c(u, v) - f(u, v)$ è positivo e $(u, v) \in E_f$. Poiché un arco (u, v) può comparire in una rete residua solo se almeno uno tra (u, v) ed (v, u) compare nella rete originale, si ha il vincolo

$$|E_f| \le 2\,|E| \ .$$

Si osservi che la rete residua G_f è essa stessa una rete di flusso con capacità date da c_f. Il lemma seguente mostra la relazione tra un flusso in una rete residua ed un flusso nella rete di flusso originale.

Lemma 27.2

Sia $G = (V, E)$ una rete di flusso con sorgente s e pozzo t e sia f un flusso in G. Sia G_f la rete residua di G indotta da f e sia f' un flusso in G_f. Allora, la somma di flussi $f + f'$ definita dall'equazione (27.4) è un flusso in G con valore $|f + f'| = |f| + |f'|$.

Dimostrazione. Dobbiamo verificare che siano soddisfatte l'antisimmetria, i vincoli di capacità e la conservazione del flusso. Per l'antisimmetria, si noti che per ogni $u, v \in V$, abbiamo

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &= -f(v, u) - f'(v, u) \\ &= -(f(v, u) + f'(v, u)) \\ &= -(f + f')(v, u) \ .\end{aligned}$$

Per i vincoli di capacità, si noti che $f'(u, v) \le c_f(u, v)$ per tutti gli $u, v \in V$. Quindi, per l'equazione (27.5),

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\le f(u, v) + (c(u, v) - f(u, v)) \\ &= c(u, v) \ .\end{aligned}$$

Per la conservazione del flusso, si noti che per tutti gli $u \in V - \{s, t\}$, abbiamo

$$\begin{aligned}\sum_{v \in V} (f + f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v)) \\ &= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) \\ &= 0 + 0 \\ &= 0 \ .\end{aligned}$$

Infine, abbiamo

$$\begin{aligned}|f + f'| &= \sum_{v \in V} (f + f')(s, v) \\ &= \sum_{v \in V} (f(s, v) + f'(s, v)) \\ &= \sum_{v \in V} f(s, v) + \sum_{v \in V} f'(s, v) \\ &= |f| + |f'| \ .\end{aligned}$$

■

Cammini aumentanti

Data una rete di flusso $G = (V, E)$ ed un flusso f, un ***cammino aumentante*** p è un cammino semplice da s a t nella rete residua G_f. Per la definizione di rete residua, ogni arco (u, v) su un cammino aumentante ammette un flusso netto positivo addizionale da u a v senza violare il vincolo di capacità sull'arco.

Il cammino grigio della figura 27.4(b) è un cammino aumentante. Trattando la rete residua G_f nella figura come una rete di flusso, possiamo inviare fino a 4 unità di flusso netto addizionale lungo ogni arco di questo cammino senza violare alcun vincolo di capacità, visto che la capacità residua più piccola su questo cammino è $c_f(v_2, v_3) = 4$. La massima quantità di flusso netto che possiamo inviare lungo gli archi di un cammino aumentante p è chiamata la ***capacità residua*** di p, ed è data da

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ è un arco di } p\}.$$

Il lemma seguente, la cui dimostrazione viene lasciata come Esercizio 27.2-7, rende più precisa l'argomentazione precedente.

Lemma 27.3

Sia $G = (V, E)$ una rete di flusso, sia f un flusso in G e sia p un cammino aumentante in G_f. Si definisca una funzione $f_p : V \times V \to \mathbb{R}$ come

$$f_p(u, v) = \begin{cases} c_f(p) & \text{se } (u, v) \text{ è su } p \ , \\ -c_f(p) & \text{se } (v, u) \text{ è su } p \ , \\ 0 & \text{altrimenti} \ . \end{cases} \tag{27.6}$$

Allora f_p è un flusso in G_f con valore $|f_p| = c_f(p) > 0$. ■

Il corollario seguente mostra che se aggiungiamo f_p ad f otteniamo un nuovo flusso in G il cui valore è più vicino al massimo. La figura 27.4(c) mostra il risultato ottenuto aggiungendo il flusso f_p di figura 27.4(b) al flusso f di figura 27.4(a).

Corollario 27.4

Siano dati una rete di flusso $G = (V, E)$, un flusso f per G ed un cammino aumentante p in G_f. Sia f_p definita come nell'equazione (27.6), e si definisca una funzione $f' : V \times V \to \mathbf{R}$ come $f' = f + f_p$. Allora f' è un flusso in G con valore $|f'| = |f| + |f_p| > |f|$.

Dimostrazione. Immediata dai lemmi 27.2 e 27.3. ■

Tagli in reti di flusso

Il metodo di Ford-Fulkerson aumenta ripetutamente il flusso lungo cammini aumentanti finché non si raggiunge un flusso massimo. Il teorema flusso massimo taglio minimo, che dimostreremo tra poco, ci dice che un flusso è massimo se e solo se la sua rete residua non contiene cammini aumentanti. Per dimostrare questo teorema dobbiamo prima esaminare la nozione di taglio in una rete di flusso.

Un ***taglio*** (S, T) di una rete di flusso $G = (V, E)$ è una partizione di V in S e $T = V - S$ tale che $s \in S$ e $t \in T$. (Questa definizione è simile alla definizione di "taglio" che abbiamo usato per gli alberi di copertura minimi nel Capitolo 24, ad eccezione del fatto che qui si tratta di grafi orientati invece che non orientati e che si richiede che $s \in S$ e $t \in T$). Se f è un flusso, allora il ***flusso netto*** attraverso il taglio (S, T) è definito come $f(S, T)$. La ***capacità*** del taglio (S, T) è $c(S, T)$.

La figura 27.5 mostra il taglio $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ nella rete di flusso di figura 27.1(b). Il flusso netto attraverso questo taglio è

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_3) + f(v_2, v_4) &= 12 + (-4) + 11 \\ &= 19 , \end{aligned}$$

e la sua capacità è

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26 . \end{aligned}$$

Si osservi che il flusso netto attraverso un taglio può comprendere flussi netti negativi tra i vertici, ma la capacità di un taglio è composta interamente da valori non negativi.

Il lemma seguente mostra che il valore di un flusso in una rete è il flusso netto attraverso un qualunque taglio della rete

Lemma 27.5

Sia f un flusso in una rete di flusso G con sorgente s e pozzo t e sia (S, T) un taglio di G. Allora, il flusso netto attraverso (S, T) è $f(S, T) = |f|$.

Figura 27.5 *Un taglio (S, T) nella rete di flusso della figura 27.1(b), dove $S = \{s, v_1, v_2\}$ e $T = \{v_3, v_4, t\}$. I vertici in S sono neri, mentre quelli in T sono bianchi. Il flusso netto attraverso (S, T) è $f(S, T) = 19$ e la capacità è $c(S, T) = 26$.*

Dimostrazione. Usando ripetutamente il lemma 27.1, abbiamo

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) \\ &= f(S, V) \\ &= f(s, V) + f(S - s, V) \\ &= f(s, V) \\ &= |f| . \end{aligned}$$

■

Un corollario immediato del lemma 27.5 è il risultato che abbiamo dimostrato nell'equazione (27.3), cioè che il valore di un flusso è il flusso netto che entra nel pozzo.

Un altro corollario del lemma 27.5 mostra come si possono usare le capacità dei tagli per limitare il valore di un flusso.

Corollario 27.6

Il valore di un qualunque flusso f in una rete di flusso G è limitato superiormente dalla capacità di un qualunque taglio di G.

Dimostrazione. Sia (S, T) un qualunque taglio di G e sia f un qualunque flusso. Per il lemma 27.5 ed i vincoli di capacità,

$$\begin{aligned} |f| &= f(S, T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u, v) \\ &\le \sum_{u \in S} \sum_{v \in T} c(u, v) \\ &= c(S, T) . \end{aligned}$$

■

Siamo ora pronti per dimostrare l'importante teorema flusso massimo taglio minimo.

Teorema 27.7 (Teorema flusso massimo taglio minimo)

Se f è un flusso in una rete di flusso $G = (V, E)$ con sorgente s e pozzo t, allora le seguenti condizioni sono equivalenti:

1. f è un flusso massimo in G.
2. La rete residua G_f non contiene cammini aumentanti.
3. $|f| = c(S, T)$ per un qualche taglio (S, T) di G.

Dimostrazione. (1) ⇒ (2): si supponga, per assurdo, che f sia un flusso massimo in G ma che G_f abbia un cammino aumentante p. Allora per il corollario 27.4, la somma dei flussi $f + f_p$, dove f è dato dall'equazione (27.6), è un flusso in G con valore strettamente maggiore di $|f|$, contraddicendo l'ipotesi che f sia un flusso massimo.

(2) ⇒ (3): si supponga che G_f non abbia cammini aumentanti, cioè, che G_f non contenga cammini da s a t. Si definisca

$S = \{v \in V$: esiste un cammino da s a v in $G\}$

e $T = V - S$. La partizione (S, T) è un taglio: ovviamente $s \in S$ ed inoltre $t \notin S$ perché non vi è nessun cammino da s a t in G_f. Per ogni coppia di vertici u e v tali che $u \in S$ e $v \in T$, abbiamo che $f(u, v) = c(u, v)$ perché altrimenti avremmo che $(u, v) \in E_f$ e v sarebbe nell'insieme S. Quindi, per il lemma 27.5 abbiamo $|f| = f(S, T) = c(S, T)$.

(3) ⇒ (1): Per il corollario 27.6, $|f| \le c(S, T)$ per tutti i tagli (S, T). La condizione $|f| = c(S, T)$ implica quindi che f sia un flusso massimo. ■

Algoritmo di Ford-Fulkerson di base

Ad ogni iterazione del metodo di Ford-Fulkerson si trova *un qualunque* cammino aumentante p e si aumenta il flusso f lungo p della capacità residua $c_f(p)$. La realizzazione del metodo che segue calcola il flusso massimo in un grafo $G = (V, E)$ aggiornando il flusso netto $f[u, v]$ tra ogni coppia u, v di vertici che sono collegati da un arco[1]. Se u e v non sono collegati da un arco in nessuna direzione, assumeremo implicitamente che $f[u, v] = 0$. Il codice assume che la capacità da u a v venga fornita in tempo costante da una funzione $c(u, v)$ con $c(u, v) = 0$ se $(u, v) \notin E$. (In una tipica realizzazione, $c(u, v)$ può essere ottenuta da campi memorizzati con i vertici e le loro liste di adiacenza). La capacità residua $c_f(u, v)$ viene calcolata secondo la formula (27.5). L'espressione $c_f(p)$ nel codice è in effetti solo una variabile temporanea che memorizza la capacità residua del cammino p.

```
FORD-FULKERSON(G, s, t)
1  for ogni arco (u, v) ∈ E[G]
2      do  f[u, v] ← 0
3          f[v, u] ← 0
4  while esiste un cammino p da s a t nella rete residua G_f
5      do  c_f(p) ← min {c_f(u, v) : (u, v) è in p}
6          for ogni arco (u, v) in p
7              do  f[u, v] ← f[u, v] + c_f(p)
8                  f[v, u] ← −f[u, v]
```

[1] Usiamo parentesi quadre quando trattiamo un identificatore (come f) come un campo variabile, mentre usiamo parentesi tonde quando lo trattiamo come una funzione.

Figura 27.6 *Un'esecuzione dell'algoritmo di Ford-Fulkerson di base.* **(a)-(d)** *Iterazioni successive del ciclo* **while**. *La parte sinistra di ogni sottofigura mostra la rete residua G_f della linea 4 con un cammino aumentante p in grigio. La parte destra mostra invece il nuovo flusso f che si ottiene aggiungendo f_p a f. La rete residua in (a) è la rete iniziale G.* **(e)** *La rete residua al momento dell'ultimo test del ciclo* **while**. *Essa non ha nessun cammino aumentante e quindi il flusso f mostrato in (d) è un flusso massimo.*

L'algoritmo FORD-FULKERSON è semplicemente un'espansione dello pseudocodice FORD-FULKERSON-METHOD presentato prima. La figura 27.6 mostra il risultato di ogni iterazione in un'esecuzione campione. Le linee 1-3 inizializzano il flusso f a 0.

Il ciclo **while** delle linee 4-8 trova ripetutamente un cammino aumentante p in G_f ed aumenta il flusso f lungo p della capacità residua $c_f(p)$. Quando non esistono più cammini aumentanti, il flusso f è un flusso massimo.

Analisi della procedura FORD-FULKERSON

Il tempo di esecuzione di FORD-FULKERSON dipende da come viene determinato il cammino aumentante p alla linea 4. Se questa scelta è fatta male, l'algoritmo può anche non terminare: il valore del flusso aumenta con incrementi successivi, ma non raggiunge necessariamente un valore di flusso massimo. Se però il cammino aumentante è scelto usando una ricerca in ampiezza (paragrafo 23.2), allora l'algoritmo richiede tempo polinomiale. Prima di dimostrare questo risultato, otterremo un semplice limite superiore per il tempo d'esecuzione nel caso in cui il cammino aumentante sia scelto in modo arbitrario e tutte le capacità siano intere.

Spesso nella pratica il problema di flusso massimo si presenta con capacità intere; inoltre, se le capacità sono numeri razionali, si può usare un'opportuna trasformazione scalare per renderle tutte intere. In questo caso, una semplice realizzazione di FORD-FULKERSON ha un tempo d'esecuzione $(E|f^*|)$, dove f^* è il flusso massimo trovato dall'algoritmo. Infatti, le linee da 1 a 3 richiedono tempo $\Theta(E)$; il ciclo **while** delle linee 4-8 viene eseguito al massimo $|f^*|$ volte perché il valore del flusso aumenta almeno di una unità in ogni iterazione.

Il lavoro all'interno del ciclo **while** può essere effettuato efficientemente se si gestisce in modo efficiente la struttura di dati usata per realizzare la rete $G = (V, E)$. Si assuma di mantenere una struttura di dati corrispondente ad un grafo orientato $G' = (V, E')$, dove $E' = \{(u, v) : (u, v) \in E \text{ oppure } (v, u) \in E\}$. Gli archi nella rete G sono anche archi in G' ed è quindi facile mantenere le capacità ed i flussi in questa struttura di dati. Dato un flusso f su G, gli archi della rete residua G_f sono gli archi (u, v) di G' tali che $c(u, v) - f[u, v] \neq 0$. Il tempo necessario per trovare un cammino in una rete residua è quindi $O(E') = O(E)$ se si usa una ricerca in profondità o in ampiezza. Ogni iterazione del ciclo **while** richiede, quindi, tempo $O(E)$ ed il tempo totale di esecuzione di FORD-FULKERSON è, di conseguenza, $O(E|f^*|)$.

Quando le capacità sono intere ed il valore di flusso massimo $|f^*|$ è piccolo, il tempo di esecuzione dell'algoritmo FORD-FULKERSON è basso. La figura 27.7(a) mostra un esempio di ciò che può accadere su una semplice rete di flusso se $|f^*|$ è grande. Un flusso massimo in questa rete ha valore 2000000 : 1000000 unità di flusso seguono il cammino $s \to u \to t$ e altre 1000000 unità di flusso seguono il cammino $s \to v \to t$. Se il primo cammino aumentante trovato da FORD-FULKERSON è $s \to u \to v \to t$, come mostrato nella figura 27.7(a), il flusso ha valore 1 dopo la prima iterazione. La rete residua risultante è mostrata nella figura 27.7(b). Se la seconda iterazione trova il cammino aumentante $s \to v \to u \to t$, come mostrato nella figura 27.7(b), allora il flusso avrà il valore di 2: la figura 27.7(c) mostra la risultante rete residua. Si può continuare in questo modo, scegliendo il cammino aumentante $s \to u \to v \to t$ nelle iterazioni dispari ed il cammino aumentante $s \to v \to u \to t$ nelle iterazioni pari. In questo modo si effettuerebbero un totale di 2000000 aumenti, incrementando il valore del flusso di una sola unità alla volta.

Il limite superiore per il tempo di esecuzione della procedura FORD-FULKERSON può essere migliorato se si realizza il calcolo del cammino aumentante p nella linea 4 con una visita in ampiezza, cioè se il cammino aumentante è un cammino *minimo* da s a t nella rete residua, dove ogni arco ha distanza (peso) unitario. Chiameremo il metodo di Ford-Fulkerson così modificato ***algoritmo di Edmonds-Karp***. Dimostriamo ora che l'algoritmo di Edmonds-Karp richiede tempo $O(VE^2)$.

Figura 27.7 (a) *Una rete di flusso per la quale* FORD-FULKERSON *può richiedere un tempo* $\Theta(E|f^*|)$, *dove* $|f^*|$ *è un flusso massimo: quella mostrata nella figura ha* $|f^*| = 2000000$. *Un cammino aumentante con capacità residua* 1 *è mostrato in grigio.* (b) *La rete residua risultante, in cui viene indicato un altro cammino aumentante con capacità* 1. (c) *La rete residua risultante.*

L'analisi si basa sulle distanze dei vertici nella rete residua G_f. Il lemma seguente usa la notazione $\delta_f(u, v)$ per denotare la distanza di cammino minimo da u a v in G_f, dove ogni arco ha distanza unitaria.

Lemma 27.8

Se l'algoritmo di Edmonds-Karp viene eseguito su una rete di flusso $G = (V, E)$ con sorgente s e pozzo t, allora, per tutti i vertici $v \in V - \{s, t\}$, la distanza di cammino minimo $\delta_f(s, v)$ nella rete residua G_f, cresce monotonamente con ogni aumento di flusso.

Dimostrazione. Si supponga per assurdo che per un qualche vertice $v \in V - \{s, t\}$ esista un aumento di flusso che faccia decrescere $\delta_f(s, v)$. Sia f il flusso subito prima dell'aumento e sia f' il flusso subito dopo. Allora

$$\delta_{f'}(s, v) < \delta_f(s, v) .$$

Si può assumere, senza perdita di generalità, che $\delta_{f'}(s, v) < \delta_{f'}(s, u)$ per tutti i vertici $u \in V - \{s, t\}$ tali che $\delta_{f'}(s, u) < \delta_f(s, u)$. Equivalentemente, possiamo assumere che, per tutti i vertici $u \in V - \{s, t\}$,

$$\delta_{f'}(s, u) < \delta_{f'}(s, v) \quad \text{implica} \quad \delta_f(s, u) \leq \delta_{f'}(s, u) . \tag{27.7}$$

Prendiamo ora un cammino minimo p' in $G_{f'}$ della forma $s \leadsto u \to v$ e consideriamo il vertice u che precede v lungo questo cammino. Allora, dobbiamo avere che $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$ per il corollario 25.2, poiché (u, v) è un arco di p' che è un cammino minimo da s a v. Per la proprietà (27.7), abbiamo quindi che

$$\delta_f(s, u) \leq \delta_{f'}(s, u) .$$

Con i vertici v e u determinati in questo modo, possiamo considerare il flusso netto f da u a v prima dell'aumento di flusso in G_f. Se $f[u, v] < c(u, v)$, allora abbiamo

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 && \text{(per il lemma 25.3)} \\ &\leq \delta_{f'}(s, u) + 1 \\ &= \delta_{f'}(s, v) , \end{aligned}$$

che contraddice l'ipotesi che l'aumento di flusso decrementi la distanza da s a v.

Quindi, si deve avere $f[u, v] = c(u, v)$, il che significa che $(u, v) \notin E_f$. Ora, il cammino aumentante p che era stato scelto in G_f per produrre $G_{f'}$ deve contenere l'arco (v, u) *nella direzione* da v a u poiché $(u, v) \in E_f$ (per ipotesi) e $(u, v) \notin E_{f'}$ come abbiamo appena mostrato. In altre parole, l'aumento di flusso lungo il cammino p invia flusso *all'indietro* lungo (u, v) e v compare prima di u in p. Poiché p è un cammino minimo da s a t, i suoi sottocammini sono anch'essi cammini minimi (lemma 25.1) e quindi abbiamo $\delta_f(s, u) = \delta_f(s, v) + 1$. Di conseguenza,

$$\begin{aligned}
\delta_{f'}(s, v) &= \delta_f(s, u) - 1 \\
&\le \delta_{f'}(s, u) - 1 \\
&= \delta_{f'}(s, v) - 2 \\
&< \delta_{f'}(s, v) ,
\end{aligned}$$

che contraddice la nostra ipotesi iniziale. ■

Il prossimo teorema fornisce un limite superiore al numero di iterazioni dell'algoritmo di Edmonds-Karp.

Teorema 27.9

Se l'algoritmo di Edmonds-Karp viene eseguito su una rete di flusso $G = (V, E)$ con sorgente s e pozzo t, allora il numero totale di aumenti di flusso effettuati dall'algoritmo è al massimo $O(VE)$.

Dimostrazione. Diciamo che un arco (u, v) in una rete residua G_f è ***critico*** su un cammino aumentante p se la capacità residua di p è la capacità residua di (u, v), cioè, se $c_f(p) = c_f(u, v)$. Dopo aver aumentato il flusso lungo un cammino aumentante, ogni arco critico lungo il cammino scompare dalla rete residua; inoltre, su ogni cammino aumentante vi è almeno un arco critico.

Siano u e v due vertici in V collegati da un arco in E. Quante volte (u, v) può diventare un arco critico durante l'esecuzione dell'algoritmo di Edmonds-Karp? Poiché i cammini aumentanti sono cammini minimi, quando (u, v) diventa critico per la prima volta abbiamo

$$\delta_f(s, v) = \delta_f(s, u) + 1 .$$

Una volta che il flusso viene aumentato, l'arco (u, v) scompare dalla rete residua. Esso non può ricomparire più tardi su di un altro cammino aumentante finché il flusso netto da u a v non decresce, e questo può accadere solo se (v, u) compare su un cammino aumentante. Se f' è il flusso in G quando si verifica questo evento, allora abbiamo

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 .$$

Poiché $\delta_f(s, v) \le \delta_{f'}(s, v)$ per il lemma 27.8, si ha che

$$\begin{aligned}
\delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\
&\ge \delta_f(s, v) + 1 \\
&= \delta_f(s, u) + 2 .
\end{aligned}$$

Di conseguenza, dal momento in cui (u, v) diventa critico al momento in cui esso diventa nuovamente critico, la distanza di u dalla sorgente aumenta almeno di 2 unità. La distanza di u dalla sorgente è, inizialmente, almeno 0 e fino al momento in cui esso diventa irraggiungibile dalla sorgente, se questo mai si verifica, la sua distanza è al massimo $|V| - 2$. Quindi, (u, v) può diventare critico al massimo $O(V)$ volte. Poiché vi sono $O(E)$ coppie di vertici che possono essere adiacenti in un grafo residuo, il numero totale di archi critici durante l'intera esecuzione dell'algoritmo di Edmonds-Karp è $O(VE)$; quindi, l'enunciato del teorema segue dall'osservazione che ogni cammino aumentante ha almeno un arco critico. ■

Poiché ogni iterazione di FORD-FULKERSON può essere realizzata in tempo $O(E)$ quando il cammino aumentante viene trovato con una visita in ampiezza, il tempo d'esecuzione totale dell'algoritmo di Edmonds-Karp è $O(VE^2)$. L'algoritmo presentato nel paragrafo 27.4 fornisce un metodo per ottenere un tempo d'esecuzione $O(V^2E)$ che costituisce la base dell'algoritmo di tempo $O(V^3)$ del paragrafo 27.5.

Esercizi

27.2-1 Nella figura 27.1(b), qual è il flusso attraverso il taglio $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? Qual è la capacità di questo taglio?

27.2-2 Si mostri l'esecuzione dell'algoritmo di Edmonds-Karp sulla rete di flusso della figura 27.1(a).

27.2-3 Nell'esempio della figura 27.6, qual è il taglio minimo che corrisponde al flusso massimo mostrato? Tra i cammini aumentanti che compaiono nell'esempio, quali sono i due cammini che annullano il flusso che era stato inviato precedentemente?

27.2-4 Si dimostri che, per ogni coppia di vertici u, v e per qualunque coppia di funzioni di capacità e flusso c ed f, si ha che $c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u)$.

27.2-5 Ricordiamo che la costruzione nel paragrafo 27.1, che converte una rete di flusso con sorgenti e pozzi multipli in una rete con sorgente e pozzo singoli, aggiunge degli archi con capacità infinita. Si dimostri che qualunque flusso nella rete risultante ha un valore finito se gli archi della rete originale hanno capacità finita.

27.2-6 Si supponga che ogni sorgente s_i in un problema con sorgenti e pozzi multipli produca esattamente p_i unità di flusso, cosicché $f(V, t_i) = q_i$. Inoltre, si supponga che ogni pozzo t_j consumi esattamente q_j unità, cosicché $f(V, t_j) = q_j$ dove $\sum_i p_i = \sum_j q_j$. Si mostri come si può trasformare il problema di trovare un flusso f che soddisfi questi vincoli addizionali nel problema di trovare un flusso massimo in una rete con sorgente e pozzo singoli.

27.2-7 Si dimostri il lemma 27.3.

27.2-8 Si mostri che un flusso massimo in una rete $G = (V, E)$ può essere sempre ricavato da una sequenza di al massimo $|E|$ cammini aumentanti. (*Suggerimento*: si determinino i cammini *dopo* aver trovato il flusso massimo).

27.2-9 La ***connettività per archi*** di un grafo non orientato è il minimo numero k di archi che devono essere rimossi per disconnettere il grafo. Ad esempio, la connettività per archi di un albero è 1, mentre la connettività per archi di una catena ciclica di archi è 2. Si mostri come si può determinare la connettività di archi di un grafo non orientato $G = (V, E)$ eseguendo un algoritmo di flusso massimo su al più $|V|$ reti di flusso, ognuna con $O(V)$ vertici ed $O(E)$ archi.

27.2-10 Si mostri che l'algoritmo di Edmonds-Karp termina dopo al più $|V|\,|E|/4$ iterazioni. (*Suggerimento*: per ogni arco (u, v), si considerino i cambiamenti di $\delta(s, u)$ e $\delta(u, t)$ tra gli istanti in cui (u, v) è critico).

27.3 Abbinamento massimo in un grafo bipartito

Alcuni problemi combinatori possono essere riformulati facilmente come problemi di flusso massimo: il problema di flusso massimo con sorgenti e pozzi multipli del paragrafo 27.1 ne è un esempio. Vi sono altri problemi combinatori che hanno, in apparenza, poco a che fare con le reti di flusso, ma che in effetti possono essere ridotti ad un problema di flusso massimo. In questo paragrafo presenteremo uno di questi problemi: trovare un abbinamento massimo in un grafo bipartito (si veda il paragrafo 5.4). Per risolvere questo problema sfrutteremo una proprietà di integrità di cui gode il metodo di Ford-Fulkerson; vedremo anche che si può risolvere il problema di abbinamento massimo bipartito su un grafo $G = (V, E)$ mediante il metodo di Ford-Fulkerson in tempo $O(VE)$.

Il problema di abbinamento massimo in un grafo bipartito

Dato un grafo non orientato $G = (V, E)$, un ***abbinamento*** è un sottoinsieme di archi $M \subseteq E$ tale che, per ogni vertice $v \in V$, al massimo un arco di M è incidente a v. Diremo che un vertice $v \in V$ è ***abbinato*** da un abbinamento M se un arco in M è incidente a v; altrimenti v è ***non abbinato***. Un ***abbinamento massimo*** è un abbinamento di cardinalità massima, cioè un abbinamento M tale che per ogni abbinamento M' abbiamo $|M| \geq |M'|$. In questo paragrafo, concentreremo la nostra attenzione sulla ricerca di abbinamenti massimi in grafi bipartiti. Si assume che l'insieme dei vertici possa essere partizionato in $V = L \cup R$, dove L ed R sono insiemi disgiunti e tutti gli archi in E vanno tra L ed R. Assumiamo, inoltre, che nessun vertice in V sia isolato. La figura 27.8 illustra la nozione di abbinamento.

Il problema di trovare un abbinamento massimo in un grafo bipartito ha diverse applicazioni pratiche. Ad esempio, si può pensare di abbinare un insieme L di macchine con un insieme R di processi che devono essere eseguiti simultaneamente: la presenza di un arco (u, v) in E significa allora che una particolare macchina $u \in L$ è capace di eseguire un particolare processo $v \in R$. In questo caso, un abbinamento massimo fornisce lavoro per il massimo numero possibile di macchine.

Figura 27.8 *Un grafo bipartito $G = (V, E)$ con partizione dei vertici $V = L \cup R$.* (**a**) *Un abbinamento con cardinalità 2.* (**b**) *Un abbinamento massimo con cardinalità 3.*

Ricerca di un abbinamento massimo bipartito

Si può usare il metodo di Ford-Fulkerson per trovare un abbinamento massimo in un grafo non orientato bipartito $G = (V, E)$ in tempo polinomiale in $|V|$ e $|E|$. Il trucco consiste nel costruire una rete di flusso in cui il flusso corrisponde agli abbinamenti, come mostrato nella figura 27.9. Dato un grafo bipartito G, la ***rete di flusso corrispondente*** $G' = (V', E')$ è definita nel modo seguente. L'insieme di vertici è $V' = V \cup \{s, t\}$, dove la sorgente s ed il pozzo t sono due vertici nuovi non in V. Se la partizione dei vertici di G è $L \cup R$, gli archi orientati di G' sono dati da

$$\begin{aligned} E' \;=\; & \{(s, u) : u \in L\} \\ & \cup \{(u, v) : u \in L,\ v \in R,\ \text{e } (u, v) \in E\} \\ & \cup \{(v, t) : v \in R\}\ . \end{aligned}$$

Per completare la costruzione, si assegna una capacità unitaria ad ogni arco in E'.

Il teorema seguente mostra che un abbinamento in G corrisponde direttamente ad un flusso nella rete di flusso G' corrispondente a G. Si dice che un flusso f su una rete di flusso $G = (V, E)$ è a ***valori interi*** se $f(u, v)$ è un intero per ogni $(u, v) \in V \times V$.

Lemma 27.10

Sia $G = (V, E)$ un grafo bipartito con partizione dei vertici $V = L \cup R$ e sia $G' = (V', E')$ la corrispondente rete di flusso. Se M è un abbinamento in G, allora esiste un flusso a valori interi f in G' con valore $|f| = |M|$. Viceversa, se f è un flusso a valori interi in G', allora esiste un abbinamento M in G con cardinalità $|M| = |f|$.

Dimostrazione. Prima mostriamo che un abbinamento M in G corrisponde ad un flusso a valori interi in G'. Si definisca f nel modo seguente: se $(u, v) \in M$, allora $f(s, u) = f(u, v) = f(v, t) = 1$ e $f(u, s) = f(v, u) = f(t, v) = -1$; per tutti gli altri archi $(u, v) \in E$ si ponga $f(u, v) = 0$.

Intuitivamente, ogni arco $(u, v) \in M$ corrisponde ad una unità di flusso in G' che attraversa il cammino $s \to u \to v \to t$. Inoltre, i cammini indotti dagli archi in M hanno vertici disgiunti, ad eccezione che per s e t. Per verificare che f soddisfi l'antisimmetria, i vincoli di capacità

Figura 27.9 *La rete di flusso corrispondente ad un grafo bipartito.* (a) *Il grafo bipartito $G = (V, E)$ di figura 27.8 con partizione dei vertici $V = L \cup R$. Un abbinamento massimo è mostrato dagli archi grigi.* (b) *La corrispondente rete di flusso G' sulla quale è indicato un flusso massimo. Ogni arco ha capacità unitaria: gli archi grigi hanno flusso di 1, mentre tutti gli altri non portano flusso. Gli archi grigi da L ad R corrispondono a quelli in un abbinamento massimo sul grafo bipartito di partenza.*

e la conservazione del flusso, è sufficiente osservare che f può essere ottenuto con un aumento di flusso lungo ognuno di questi cammini. Il flusso netto attraverso il taglio $(L \cup \{s\}, R \cup \{t\})$ è uguale a $|M|$; quindi, per il lemma 27.5, il valore del flusso è $|f| = |M|$.

Per dimostrare il viceversa, sia f un flusso a valori interi in G' e sia

$$M = \{(u, v) : u \in L,\ v \in R,\ \text{e}\ f(u, v) > 0\}\ .$$

Ogni vertice $u \in L$ ha un solo arco entrante, cioè (s, u), e la sua capacità è 1; quindi, ogni $u \in L$ ha al massimo un'unità di flusso netto positivo entrante in esso. Poiché f è a valori interi, per ogni $u \in L$, abbiamo che un'unità di flusso netto positivo entra in u se e solo se vi è esattamente un vertice $v \in R$ tale che $f(u, v) = 1$; quindi, al massimo un arco uscente da u può portare un flusso netto positivo. Poiché un argomento simmetrico può essere portato per ogni $v \in R$ abbiamo che l'insieme M definito nell'enunciato del teorema è effettivamente un abbinamento.

Per vedere che $|M| = |f|$ si osservi che, per ogni vertice abbinato $u \in L$, abbiamo $f(s, u) = 1$ e, per ogni arco $(u, v) \in E - M$, abbiamo $f(u, v) = 0$. Di conseguenza, usando il lemma 27.1, l'antisimmetria ed il fatto che non vi sono archi da L a t, otteniamo

$$\begin{aligned} |M| &= f(L, R) \\ &= f(L, V') - f(L, L) - f(L, s) - f(L, t) \\ &= 0 - 0 - f(s, L) - 0 \\ &= f(s, V') \\ &= |f|\ . \end{aligned}$$

È un fatto intuitivo che un abbinamento massimo in un grafo bipartito G corrisponda ad un flusso massimo nella rete di flusso corrispondente G'. Quindi, possiamo calcolare un abbinamento massimo in G eseguendo un algoritmo di flusso massimo su G'. L'unico problema in questo ragionamento è che l'algoritmo di flusso massimo potrebbe restituire un flusso in G' che contiene quantità non intere; il seguente teorema mostra, però, che se usiamo il metodo di Ford-Fulkerson questa eventualità non può mai verificarsi.

Teorema 27.11 (Teorema di integrità)

Se la funzione capacità c assume solo valori interi, allora il flusso massimo f prodotto dal metodo Ford-Fulkerson gode della proprietà che $|f|$ è un valore intero. Inoltre, per ogni coppia di vertici u e v, il valore di $f(u, v)$ è un intero.

Dimostrazione. La dimostrazione è per induzione sul numero di iterazioni: la lasciamo al lettore come Esercizio 27.3-2. ■

Possiamo ora dimostrare il seguente corollario del lemma 27.10.

Corollario 27.12

La cardinalità di un abbinamento massimo in un grafo bipartito G è il valore di un flusso massimo nella rete di flusso corrispondente G'.

Dimostrazione. Usando i simboli introdotti nel lemma 27.10, si supponga che M sia un abbinamento massimo in G e che il corrispondente flusso f in G' non sia massimo. Allora esiste un flusso massimo f' in G tale che $|f'| > |f|$. Poiché le capacità in G' hanno valori interi, per il Teorema 27.11 anche f' ha valori interi; quindi, f' corrisponde ad un abbinamento M' in C con cardinalità $|M'| = |f'| > |f| = |M|$ e questo contraddice il fatto che M è massimo. In modo analogo, si può mostrare che se f' è un flusso massimo in corrispondente abbinamento è un abbinamento massimo su G. ■

Quindi, dato un grafo non orientato e bipartito G, possiamo trovare un abbinamento massimo creando la rete di flusso G', eseguendo il metodo di Ford-Fulkerson ed ottenendo direttamente l'abbinamento massimo M dal flusso massimo a valori interi f così trovato. Poiché ogni abbinamento in un grafo bipartito ha cardinalità al massimo $\min(L, R) = O(V)$, il valore del flusso massimo in G' è $O(V)$; di conseguenza, si può trovare un abbinamento massimo in un grafo bipartito in tempo $O(VE)$.

Esercizi

27.3-1 Si esegua l'algoritmo di Ford-Fulkerson sulla rete di flusso della figura 27.9(b) e si mostri la rete residua dopo ogni aumento di flusso. Si numerino i vertici in L dall'alto verso il basso da 1 a 5 e quelli in R in modo analogo da 6 a 9. Per ogni iterazione, si scelga il cammino aumentante più piccolo in ordine lessicografico.

27.3-2 Si dimostri il Teorema 27.11.

27.3-3 Sia $G = (V, E)$ un grafo bipartito con partizione dei vertici $V = L \cup R$ e sia G' la rete di flusso corrispondente. Si dia un buon limite superiore alla lunghezza di un qualunque cammino aumentante trovato in G' durante l'esecuzione di FORD-FULKERSON.

* *27.3-4* Un ***abbinamento perfetto*** è un abbinamento in cui ogni vertice è abbinato. Sia $G = (V, E)$ un grafo non orientato e bipartito con partizione dei vertici $V = L \cup R$, dove $|L| = |R|$. Per ogni $X \subseteq V$, si definisca l'***intorno*** di X come

$$N(X) = \{y \in V : (x, y) \in E \quad \text{per qualche} \quad x \in X\} ,$$

cioè l'insieme di tutti i vertici adiacenti a qualche elemento di X. Si dimostri il ***teorema di Hall***: esiste un abbinamento perfetto in G se e solo se $|A| \le |N(A)|$ per ogni sottoinsieme $A \subseteq L$.

* *27.3-5* Un grafo bipartito $G = (V, E)$, dove $V = L \cup R$, è ***d-regolare*** se ogni vertice $v \in V$ ha grado esattamente d. Ogni grafo bipartito d-regolare ha $|L| = |R|$. Si dimostri che ogni grafo bipartito d-regolare ha un abbinamento di cardinalità $|L|$ mostrando che un taglio minimo della rete di flusso corrispondente ha capacità $|L|$.

* 27.4 Algoritmi di preflusso

In questo paragrafo presentiamo l'approccio dei "preflussi" per calcolare flussi massimi. I più veloci algoritmi di flusso massimo sono, a tutt'oggi, algoritmi di preflusso; altri problemi di flusso, come il problema di flusso di costo minimo, possono essere risolti efficientemente col metodo dei preflussi. Questo paragrafo introduce l'algoritmo di flusso massimo "generico" di Goldberg che ha una semplice realizzazione che richiede tempo $O(V^2E)$, migliorando quindi il limite $O(VE^2)$ dell'algoritmo di Edmonds-Karp. Nel paragrafo 27.5, raffineremo l'algoritmo generico ottenendo un altro algoritmo di preflusso che richiede tempo $O(V^3)$.

Gli algoritmi di preflusso lavorano in modo più localizzato rispetto al metodo di Ford-Fulkerson. Invece di esaminare l'intera rete residua $G = (V, E)$ per trovare un cammino aumentante, gli algoritmi di preflusso lavorano su un vertice alla volta, considerando solo i vicini del vertice nella rete residua. Inoltre, diversamente dal metodo di Ford-Fulkerson, gli algoritmi di preflusso non mantengono la proprietà di conservazione del flusso durante la loro esecuzione: tuttavia, essi mantengono un ***preflusso*** che è una funzione $f : V \times V \to \mathbf{R}$ che soddisfa l'antisimmetria, i vincoli di capacità ed il seguente rilassamento della legge di conservazione del flusso: $f(V, u) \ge 0$ per tutti i vertici $u \in V - \{s\}$. Quindi, il flusso netto entrante in ciascun vertice diverso dalla sorgente è non negativo. Il flusso netto in un vertice u è chiamato il ***flusso in eccesso*** in u ed è dato da

$$e(u) = f(V, u) . \tag{27.8}$$

Si dice che un vertice $u \in V - \{s, t\}$ è ***traboccante*** se $e(u) > 0$.

Cominceremo questo paragrafo descrivendo l'intuizione che guida il metodo dei preflussi. Quindi, esamineremo le due operazioni che vengono utilizzate dal metodo: "inviare" un preflusso ed "innalzare" un vertice. Infine, presenteremo un algoritmo generico di preflusso e ne analizzeremo la correttezza ed il tempo d'esecuzione.

Intuizione

L'intuizione su cui si basa il metodo dei preflussi può essere compresa meglio in termini di flussi di liquidi: consideriamo una rete di flusso $G = (V, E)$ come un sistema interconnesso di tubi di prefissata capacità. Applicando questa analogia al metodo di Ford-Fulkerson, si può dire che ogni cammino aumentante nella rete produce un flusso addizionale di liquido, senza

punti di biforcazione, che scorre dalla sorgente al pozzo. Il metodo di Ford-Fulkerson aggiunge ripetutamente, finché possibile, flussi di liquido di questo tipo.

L'algoritmo generico di preflusso è basato su un'intuizione alquanto diversa. Come prima, gli archi orientati corrispondono a tubi, ma i vertici, che sono i punti di giunzione dei tubi, hanno due interessanti proprietà. La prima è che per accomodare il flusso in eccesso, ogni vertice ha un tubo di scolo che finisce in un serbatoio arbitrariamente grande dove il fluido può essere accumulato; la seconda è che ogni vertice, il suo serbatoio e tutti i tubi ad esso collegati giacciono su una piattaforma la cui altezza cresce con il progredire dell'algoritmo.

L'altezza di un vertice determina come il flusso viene inviato: infatti, il flusso viene inviato solo verso il basso, cioè da un vertice più alto ad uno più basso. Ci può essere un flusso netto positivo da un vertice più basso ad uno più alto, ma le operazioni che inviano del flusso lo fanno sempre verso il basso. L'altezza della sorgente è fissata a $|V|$, mentre quella del pozzo è fissata a 0; l'altezza di tutti gli altri vertici parte da 0 e aumenta col tempo. L'algoritmo dapprima invia quanto più flusso possibile verso il basso, dalla sorgente verso il pozzo: la quantità che viene inviata è esattamente quanto basta per riempire fino alla capacità massima ogni tubo che parte dalla sorgente: viene, cioè, inviata la capacità del taglio $(s, V - s)$. Quando il flusso raggiunge per la prima volta un vertice intermedio, esso si raccoglie nel serbatoio del vertice: da qui esso verrà inviato, prima o poi, verso il basso.

Può succedere che i soli tubi che escono da un vertice u e che non sono ancora saturi colleghino u a vertici che sono sullo stesso livello di u o più in alto. In questo caso, per liberare un vertice traboccante u dal suo flusso in eccesso, occorre aumentarne l'altezza: un'operazione chiamata "innalzamento" del vertice u. La sua altezza viene aumentata sino ad una unità più del più basso dei suoi vicini verso il quale ha un tubo non saturo. Quindi, dopo che il vertice è stato innalzato, vi è almeno un tubo uscente da esso attraverso cui può venire inviato del flusso addizionale.

Al termine, tutto il flusso che potenzialmente può attraversare la rete fino al pozzo sarà arrivato lì: di più non ne può arrivare perché i tubi rispettano i vincoli di capacità, quindi la quantità di flusso che attraversa ogni taglio è ancora limitata dalla capacità del taglio stesso. Per trasformare un preflusso in un flusso "legale", l'algoritmo rimanda verso la sorgente il fluido in eccesso raccolto nei serbatoi dei vertici traboccanti: questo viene ottenuto continuando ad innalzare i vertici oltre $|V|$, cioè oltre l'altezza prefissata della sorgente. (L'invio del flusso in eccesso all'indietro verso la sorgente viene di fatto realizzato cancellando i flussi che causano l'eccesso). Come vedremo, una volta che tutti i serbatoi sono stati svuotati, il preflusso non solo è un flusso "legale", ma è anche un flusso massimo.

Operazioni di base

Dalla discussione precedente risulta che vi sono due operazioni di base effettuate da un algoritmo di preflusso: l'invio di eccesso di flusso da un vertice ad uno dei suoi vicini, e l'innalzamento di un vertice. L'applicabilità di queste operazioni dipende dalle altezze dei vertici, che definiremo ora con precisione.

Sia $G = (V, E)$ una rete di flusso con sorgente s e pozzo t e sia f un preflusso in G. Una funzione $h : V \to \mathbf{N}$ è una ***funzione altezza*** se $h(s) = |V|$, $h(t) = 0$ e

$$h(u) \le h(v) + 1.$$

per ogni arco residuo $(u, v) \in E_f$. Da questa definizione deriva immediatamente il seguente lemma.

Lemma 27.13

Sia $G = (V, E)$ una rete di flusso, sia f un preflusso in G, e sia h una funzione altezza su V. Per ogni coppia di vertici $u, v \in V$, se $h(u) > h(v) + 1$, allora (u, v) non è un arco nel grafo residuo. ■

L'operazione di base PUSH(u, v) può essere applicata se u è un vertice traboccante, $c_f(u, v) > 0$ e $h(u) = h(v) + 1$. Lo pseudocodice riportato di seguito aggiorna il preflusso f in una rete $G = (V, E)$ implicitamente fissata. Esso assume che le capacità siano date da un funzione a tempo costante c e che anche le capacità residue possano essere calcolate in tempo costante sulla base di c e f. Il flusso in eccesso accumulato in un vertice u viene memorizzato come $e[u]$ e l'altezza di u viene mantenuta come $h[u]$. L'espressione $d_f(u, v)$ è una variabile temporanea che memorizza la quantità di flusso che può essere inviato da u a v.

PUSH(u, V)

1 ▷ **Si applica quando:** u è traboccante, $c_f(u, v) > 0$ e $h[u] = h[v] + 1$.
2 ▷ **Azione:** invia $d_f(u, v) = \min(e[u], c_f(u, v))$ unità di flusso da u a 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 $e[v] \leftarrow e[v] + d_f(u, v)$

Il codice per PUSH opera nel modo seguente. Si assume che il vertice u abbia un eccesso positivo $e[u]$ e che la capacità residua di (u, v) sia positiva. Quindi, si può inviare fino a $d_f(u, v) = \min(e[u], c_f(u, v))$ unità di flusso da u a v senza rischiare di rendere $e[u]$ negativo o di superare la capacità $c(u, v)$: questo valore viene calcolato alla linea 3. Il flusso viene mosso da u a v aggiornando f alle linee 4-5 ed e alle linee 6-7. Quindi, se f era un preflusso prima che PUSH venisse invocata, allora continua ad essere un preflusso.

Si osservi che niente nel codice di PUSH dipende dalle altezze di u e v; tuttavia PUSH non viene invocata qualora $h[u] = h[v] + 1$. Quindi, il flusso in eccesso può essere spinto verso il basso solo da una differenza di altezza di 1. Per il lemma 27.13 non esiste alcun arco residuo tra due vertici le cui altezze differiscano più di 1, quindi non ci sarebbe nulla da guadagnare permettendo al flusso di essere spinto verso il basso da una differenza di altezza maggiore di 1.

Chiameremo l'operazione PUSH(u, V) un ***invio*** da u a v; se una operazione di invio si applica ad un arco (u, v) uscente da un vertice u, diremo anche che l'operazione di invio si applica a u. Una tale operazione è un ***invio saturante*** se l'arco (u, v) diviene ***saturo*** (cioè, se dopo l'operazione $c_f(u, v) = 0$) altrimenti è un ***invio non saturante***. Se un arco è saturo, esso non compare nella rete residua.

L'operazione di base LIFT(u) si applica se u è traboccante e se $c_f(u, V) > 0$ implica $h[u] \leq h[v]$, per tutti i vertici v. In altre parole, possiamo innalzare un vertice traboccante u se per ogni vertice v per il quale esiste una capacità residua da u a v, non si può inviare del flusso da u a v perché v non è più basso di u. (Si ricordi che, per definizione, né la sorgente s né il pozzo t possono essere traboccanti e quindi né s né t possono essere innalzati.)

LIFT(u)

1 ▷ **Si applica quando:** u è traboccante e per ogni $v \in V$,
$(u, V) \in E_f$ implica $h[u] \leq h[v]$.
2 ▷ **Azione:** aumenta l'altezza di u
3 $h[u] \leftarrow 1 + \min\{h[v] : (u, v) \in E_f\}$

Quando invochiamo l'operazione LIFT(u), diciamo che il vertice u viene ***innalzato***. È importante notare che quando u viene innalzato, E_f deve contenere almeno un arco uscente da u affinché l'operazione di minimo nel codice venga applicata a un insieme non vuoto: questo fatto viene garantito dall'ipotesi che u sia traboccante. Infatti, poiché $e[u] > 0$, abbiamo che $e[u] = f[V, u] > 0$ e quindi deve esistere almeno un vertice v tale che $f[v, u] > 0$. Ma allora

$$\begin{aligned} c_f(u, v) &= c(u, v) - f[u, v] \\ &= c(u, v) + f[v, u] \\ &> 0 , \end{aligned}$$

il che implica che $(u, v) \in E_f$. L'operazione LIFT(u) attribuisce, quindi, a u l'altezza massima consentita dai vincoli sulle funzioni altezza.

Algoritmo generico

L'algoritmo generico del preflusso usa la seguente sottoprocedura per creare un preflusso iniziale nella rete di flusso.

INITIALIZE-PREFLOW(G, s)

1 **for** ogni vertice $u \in V[G]$
2 **do** $h[u] \leftarrow 0$
3 $e[u] \leftarrow 0$
4 **for** ogni arco $(u, v) \in E[G]$
5 **do** $f[u, v] \leftarrow 0$
6 $f[v, u] \leftarrow 0$
7 $h[s] \leftarrow |V[G]|$
8 **for** ogni vertice $u \in Adj[s]$
9 **do** $f[s, u] \leftarrow c(s, u)$
10 $f[u, s] \leftarrow -c(s, u)$
11 $e[u] \leftarrow c(s, u)$

La procedura INITIALIZE-PREFLOW crea un preflusso iniziale f definito da

$$f[u, v] = \begin{cases} c(u, v) & \text{se } u = s , \\ -c(v, u) & \text{se } v = s , \\ 0 & \text{altrimenti} . \end{cases} \tag{27.9}$$

Quindi, ogni arco uscente dalla sorgente viene completamente riempito, mentre tutti gli altri archi non trasportano alcun flusso. Per ogni vertice v adiacente alla sorgente, si ha all'inizio $e[v] = c(s, v)$. L'algoritmo generico comincia con una funzione altezza iniziale h, data da

$$h[u] = \begin{cases} |V| & \text{se } u = s, \\ 0 & \text{altrimenti.} \end{cases}$$

Tale funzione è un'altezza, in quanto gli unici archi (u, v) per i quali $h[u] > h[v] + 1$ sono quelli per cui $u = s$ e questi archi sono saturi, il che implica che essi non fanno parte della rete residua.

L'algoritmo seguente è un tipico esempio del metodo dei preflussi.

```
GENERIC-PREFLOW-PUSH(G)
INITIALIZE-PREFLOW(G, s)
while esiste un'operazione di invio o di innalzamento applicabile
        do seleziona un'operazione di invio o di innalzamento applicabile
                ed eseguila
```

Dopo aver inizializzato il flusso, l'algoritmo generico applica ripetutamente, in un ordine qualunque, le operazioni di base, fermandosi quando nessuna è più applicabile. Il lemma seguente ci dice che finché esiste un vertice traboccante, almeno una delle due operazioni può essere applicata.

Lemma 27.14 ***(Ad un vertice traboccante può essere applicata un'operazione di base)***

Sia $G = (V, E)$ una rete di flusso con sorgente s e pozzo t, sia f un preflusso e sia h la funzione altezza per f. Se u è un vertice traboccante, allora ad u può essere applicata un'operazione di invio o un'operazione di innalzamento.

Dimostrazione. Per ogni arco residuo (u, v), abbiamo che $h(u) \le h(v) + 1$ perché h è una funzione altezza. Se ad u non può essere applicata un'operazione di invio, allora per tutti gli archi residui (u, v) dobbiamo avere che $h(u) < h(v) + 1$ e questo implica $h(u) \le h(v)$. Quindi, si può applicare un'operazione di innalzamento ad u. ■

Correttezza del metodo dei preflussi

Per dimostrare che l'algoritmo generico del preflusso risolve il problema di flusso massimo, mostreremo prima che se esso termina allora il preflusso f è un flusso massimo; in seguito mostreremo che esso necessariamente termina. Cominciamo con alcune osservazioni sulla funzione di altezza h.

Lemma 27.15 ***(Le altezze dei vertici non decrescono mai)***

Durante l'esecuzione di GENERIC-PREFLOW-PUSH su di una rete di flusso $G = (V, E)$, per ogni vertice $u \in V$, l'altezza $h[u]$ non decresce mai. Inoltre, quando si applica un'operazione di innalzamento ad un vertice u, la sua altezza $h[u]$ cresce almeno di 1.

Dimostrazione. Poiché le altezze dei vertici possono cambiare solo durante le operazioni di innalzamento, è sufficiente dimostrare la seconda parte del lemma. Se un vertice u viene innalzato, allora per tutti i vertici v tali che $(u, v) \in E_f$ abbiamo $h[u] < h[v]$; questo implica che $h[u] \le 1 + \min\{h[v] : (u, v) \in E_f\}$ e quindi l'operazione deve necessariamente aumentare $h[u]$. ■

Lemma 27.16

Sia $G = (V, E)$ una rete di flusso con sorgente s e pozzo t. Allora durante l'esecuzione di GENERIC-PREFLOW-PUSH su G, l'attributo h contiene sempre una funzione altezza.

Dimostrazione. La dimostrazione è per induzione sul numero di operazioni base eseguite. All'inizio h è una funzione altezza, come abbiamo già osservato.

Mostriamo che se h è una funzione altezza, allora, dopo l'operazione LIFT(u), h sarà ancora una funzione altezza. Se consideriamo l'arco residuo $(u, v) \in E_f$ che esce da u, allora l'operazione LIFT(u) garantisce che, al termine, si abbia $h[u] \ge h[v] + 1$. Ora, si consideri un arco residuo (w, u) entrante in u: per il lemma 27.15, se prima dell'operazione LIFT(u) vale $h[w] \le h[u] + 1$, allora dopo l'operazione avremo $h[w] < h[u] + 1$. Quindi, dopo l'operazione LIFT(u), h rimane una funzione altezza.

Consideriamo ora una operazione PUSH(u, v). Questa operazione può aggiungere un arco (v, u) ad E_f e può rimuovere (u, v) da E_f. Nel primo caso, abbiamo che $h[v] = h[u] - 1$, quindi h rimane una funzione altezza; nel secondo, la rimozione di (u, v) dalla rete residua elimina il corrispondente vincolo e quindi anche in questo caso h rimane una funzione altezza. ■

Il seguente lemma ci fornisce una importante proprietà delle funzioni altezza.

Lemma 27.17

Sia $G = (V, E)$ una rete di flusso con sorgente s e pozzo t, sia f un preflusso in G e sia h una funzione altezza su V. Allora, non esiste alcun cammino dalla sorgente s al pozzo t nella rete residua G_f.

Dimostrazione. Si supponga per assurdo che esista un cammino $p = \langle v_0, v_1, \ldots, v_k \rangle$ da s a t in G_f, dove $v_0 = v_k$. Senza perdita di generalità, possiamo supporre che p sia semplice e quindi che $k < |V|$. Per $i = 0, 1, \ldots, k$, abbiamo che l'arco $(v_i, v_{i+1}) \in E$. Poiché h è una funzione altezza, $h(v_i) \le h(v_{i+1}) + 1$ per $i = 1, 2, \ldots, k - 1$. Combinando queste disuguaglianze lungo il cammino p otteniamo $h(s) \le h(t) + k$. Ma poiché $h(t) = 0$, abbiamo $h(s) \le k < |V|$. Il che contraddice il requisito che $h(s) = |V|$ per una funzione altezza. ■

Siamo ora pronti a dimostrare che se l'algoritmo generico di preflusso termina, allora il preflusso che esso calcola è un flusso massimo.

Teorema 27.18 ***(Correttezza dell'algoritmo generico di preflusso)***

Se l'algoritmo GENERIC-PREFLOW-PUSH termina quando è eseguito su una rete di flusso $G = (V, E)$ con sorgente s e pozzo t, allora il preflusso f che esso calcola è un flusso massimo per G.

Dimostrazione. Se l'algoritmo generico termina, allora ogni vertice in $V - \{s, t\}$ deve avere un eccesso di 0 perché, per i lemma 27.14 e 27.16 e per il fatto che f è sempre un preflusso,

si ha che non esistono vertici traboccanti: quindi f è un flusso. Inoltre, poiché h è una funzione altezza, per il lemma 27.17 non vi è alcun cammino da s a t nella rete residua G_f e quindi, per il teorema flusso massimo taglio minimo, f è un flusso massimo.

Analisi del metodo dei preflussi

Per mostrare che l'algoritmo generico del preflusso effettivamente termina, dobbiamo porre un limite superiore al numero delle operazioni che esso può eseguire. Per ognuno dei tre tipi di operazioni (cioè innalzamenti, invii saturanti ed invii non saturanti) porremo un limite indipendente. Sfruttando poi questi limiti superiori, sarà facile costruire un algoritmo che richiede tempo $O(V^2E)$. Prima di cominciare l'analisi, dimostriamo il seguente importante lemma.

Lemma 27.19

Sia $G = (V, E)$ una rete di flusso con sorgente s e pozzo t e sia f un preflusso in G. Allora, per ogni vertice traboccante u, vi è un cammino semplice da u a s nella rete residua G_f.

Dimostrazione. Sia $U = \{v : \text{esiste un cammino semplice da } u \text{ a } v \text{ in } G_f\}$ e si supponga, per assurdo, che $s \in U$. Sia $\bar{U} = V - U$.

Mostriamo che, per ogni coppia di vertici $v \in U$ e $w \in \bar{U}$, si ha $f(w, v) < 0$. Infatti, se $f(w, v) > 0$, allora $f(v, w) < 0$ il che implica che $c_f(v, w) = c(v, w) - f(v, w) > 0$. Quindi, esiste un arco $(v, w) \in E_f$ della forma $u \rightsquigarrow v \rightarrow w$ che può essere usato per formare un cammino semplice in G_f, contraddicendo il fatto che $w \in \bar{U}$.

Di conseguenza dobbiamo avere $f(\bar{U}, U) \le 0$, poiché ogni termine in questa sommatoria implicita è non positivo. Allora, dall'equazione (27.8) e dal lemma 27.1, possiamo concludere che

$$\begin{aligned} e(U) &= f(V, U) \\ &= f(\bar{U}, U) + f(U, U) \\ &= f(\bar{U}, U) \\ &\le 0 . \end{aligned}$$

Gli eccessi sono non negativi per tutti i vertici in $V - \{s\}$; poiché abbiamo assunto che $U \subseteq V - \{s\}$, dobbiamo avere $e(v) = 0$, per tutti i vertici $v \in U$. In particolare, abbiamo che $e(u) = 0$, il che contraddice l'ipotesi che u sia traboccante. ■

Il prossimo lemma stabilisce un limite superiore all'altezza dei vertici, ed il suo corollario pone un limite al numero totale di operazioni d'innalzamento che possono essere eseguite.

Lemma 27.20

Sia $G = (V, E)$ una rete di flusso con sorgente s e pozzo t. Ad ogni istante durante l'esecuzione di GENERIC-PREFLOW-PUSH su G abbiamo che $h[u] \le 2|V| - 1$ per tutti i vertici $u \in V$.

Dimostrazione. L'altezza della sorgente s e del pozzo t non cambia mai perché questi vertici sono per definizione non traboccanti: quindi, si ha sempre che $h[s] = |V|$ e $h[t] = 0$.

Poiché un vertice viene innalzato solo quando è traboccante, possiamo considerare un arbitrario vertice traboccante $u \in V - \{s, t\}$. Il lemma 27.19 ci dice che esiste un cammino semplice p da u a s in G_f; sia $p = \langle v_0, v_1, \ldots, v_k \rangle$, dove $v_0 = u$, $v_k = s$ e $k \le |V| - 1$, perché p è semplice. Per $i = 0, 1, \ldots, k - 1$, abbiamo che $(v_i, v_{i+1}) \in E_f$ e quindi, per il lemma 27.16, $h[v_i] \le h[v_{i+1}] + 1$. Espandendo queste disuguaglianze lungo il cammino p, si ottiene $h[u] = h[v_0] \le h[v_k] + k \le h[s] + (|V| - 1) = 2|V| - 1$. ■

Corollario 27.21 (*Limite superiore alle operazioni di innalzamento*)

Sia $G = (V, E)$ una rete di flusso con sorgente s e pozzo t. Allora durante l'esecuzione di GENERIC-PREFLOW-PUSH su G, il numero di operazioni di innalzamento è al massimo $2|V| - 1$ per ogni vertice, ed al massimo $(2|V| - 2)(|V| - 2) < 2|V|^2$ in totale.

Dimostrazione. Solo i vertici in $V - \{s, t\}$, che sono in tutto $|V| - 2$, possono essere innalzati. Sia ora $u \in V - \{s, t\}$: l'operazione LIFT(u) incrementa $h[u]$, il cui valore è inizialmente 0 e non può superare $2|V| - 1$ per il lemma 27.20. Quindi ogni vertice $u \in V - \{s, t\}$ può essere innalzato al massimo $2|V| - 1$ volte, ed il numero totale di operazioni di innalzamento eseguite è al massimo $(2|V| - 1)(|V| - 2) < 2|V|^2$. ■

Il lemma 27.20 è anche d'aiuto per porre un limite superiore al numero di invii saturanti.

Lemma 27.22 (*Limite superiore al numero di invii saturanti*)

Durante l'esecuzione di GENERIC-PREFLOW-PUSH su una rete di flusso $G = (V, E)$, il numero di invii saturanti è al massimo $2|V|\,|E|$.

Dimostrazione. Per ogni coppia di vertici $u, v \in V$, si considerino gli invii saturanti da u a v e da v a u. Se vi è almeno uno di questi invii, allora almeno uno tra (u, v) e (v, u) è effettivamente un arco di E. Si supponga che sia stato eseguito un invio saturante da u a v. Affinché possa essere effettuato un altro invio da u a v, l'algoritmo deve prima inviare del flusso da v a u, il che non può accadere prima che $h[v]$ cresca almeno di 2. Analogamente $h[u]$ deve crescere almeno di 2 tra due invii saturanti da v a u.

Si consideri la sequenza A degli interi dati da $h[u] + h[v]$ per ogni invio saturante tra i vertici u e v: vogliamo trovare un limite superiore alla lunghezza di questa sequenza. Quando avviene il primo invio in una qualsiasi direzione tra u e v, dobbiamo avere $h[u] + h[v] \ge 1$: quindi il primo intero in A è almeno 1. Quando avviene l'ultimo invio tra u e v, per il lemma 27.20, abbiamo che $h[u] + h[v] \le (2|V| - 1) + (2|V| - 2) = 4|V| - 3$: quindi, l'ultimo intero in A è al massimo $4|V| - 3$. Inoltre, per le considerazioni fatte sopra, al massimo un intero ogni due può comparire in A; così, il numero di interi in A è al massimo $((4|V| - 3) - 1)/2 + 1 = 2|V| - 1$ (dove abbiamo aggiunto 1 per garantire che entrambi gli estremi della sequenza vengano contati). Il numero totale di invii saturanti tra i vertici u e v è quindi al massimo $2|V| - 1$. Moltiplicando questo numero per il numero degli archi otteniamo che il numero totale di invii saturanti è al massimo $(2|V| - 1)\,|E| < 2|V|\,|E|$. ■

Il seguente lemma pone un limite superiore al numero di invii non saturanti eseguiti nell'algoritmo generico di preflusso.

Lemma 27.23 (*Limite superiore al numero di invii non saturanti*)

Durante l'esecuzione di GENERIC-PREFLOW-PUSH su una rete di flusso $G = (V, E)$, il numero totale di invii non saturanti è al massimo $4|V|^2(|V| + |E|)$.

Dimostrazione. Si definisca la funzione potenziale $\Phi = \sum_{v \in X} h[v]$ dove $X \subseteq V$ è l'insieme dei vertici traboccanti: ovviamente, all'inizio abbiamo che $\Phi = 0$. Si osservi che l'innalzamento di un vertice u incrementa Φ al massimo di $2|V|$ poiché l'insieme su cui la sommatoria viene fatta è lo stesso ed u non può essere innalzato oltre la sua massima altezza che, per il lemma 27.20, è al più $2|V|$. Inoltre, un invio saturante da un vertice u ad un vertice v aumenta Φ al massimo di $2|V|$ perché le altezze non cambiano e solo il vertice v, la cui altezza è al massimo $2|V|$, può eventualmente diventare traboccante. Infine, si osservi che un invio non saturante da u a v decrementa Φ almeno di 1 poiché u non è più traboccante dopo l'invio, v diventa invece traboccante anche se prima non lo era e $h[v] - h[u] = -1$.

Quindi, durante l'esecuzione dell'algoritmo, il numero totale di incrementi di Φ è limitato, dal corollario 27.21 e dal lemma 27.22, ad essere al massimo $(2|V|)(2|V|^2) + (2|V|)(2|V||E|) = 4|V|^2 (|V| + |E|)$. Poiché $\Phi \geq 0$, il numero totale di decrementi, e quindi il numero totale di invii non saturanti, è al massimo $4|V|^2 (|V| + |E|)$. ■

Abbiamo ora tutti gli elementi per presentare la seguente analisi della procedura GENERIC-PREFLOW-PUSH e quindi di un qualunque algoritmo basato sul metodo dei preflussi.

Teorema 27.24

Durante l'esecuzione della procedura GENERIC-PREFLOW-PUSH su una qualunque rete di flusso $G = (V, E)$, il numero di operazioni di base è $O(V^2E)$.

Dimostrazione. Immediata, per il corollario 27.21 ed i lemmi 27.22 e 27.23. ■

Corollario 27.25

Esiste una realizzazione dell'algoritmo generico del preflusso che richiede tempo $O(V^2E)$ su una qualunque rete di flusso $G = (V, E)$.

Dimostrazione. L'Esercizio 27.4-1 chiede al lettore di mostrare come si può realizzare l'algoritmo generico con un tempo addizionale di $O(V)$ per ogni operazione di innalzamento e di $O(1)$ per ogni invio. Da questo segue immediatamente l'enunciato del corollario. ■

Esercizi

27.4-1 Si mostri come si può realizzare l'algoritmo generico del preflusso usando un tempo $O(V)$ per ogni operazione di innalzamento ed un tempo $O(1)$ per ogni invio, per un tempo totale $O(V^2E)$.

27.4-2 Si dimostri che l'algoritmo generico del preflusso impiega in totale tempo $O(VE)$ per eseguire tutte le $O(V^2)$ operazioni di innalzamento.

27.4-3 Si supponga di aver trovato un flusso massimo di una rete di flusso $G = (V, E)$ usando un algoritmo di preflusso. Si dia un algoritmo efficiente per trovare un taglio minimo in G.

27.4-4 Si dia un algoritmo di preflusso efficiente per trovare un abbinamento massimo in un grafo bipartito. Si analizzi l'algoritmo proposto.

27.4-5 Si supponga che tutte le capacità degli archi di una rete di flusso $G = (V, E)$ siano nell'insieme $\{1, 2, \ldots, k\}$. Si analizzi il tempo di esecuzione dell'algoritmo generico del preflusso in termini di $|V|$, $|E|$ e k. (*Suggerimento*: quante volte ogni arco può subire un invio non saturante prima di divenire saturo?)

27.4-6 Si mostri che la linea 7 di INITIALIZE-PREFLOW può essere cambiata in

$$h[s] = |V[G]| - 2$$

senza danneggiare la correttezza o l'efficienza asintotica dell'algoritmo generico del preflusso.

27.4-7 Sia $\delta_f(u, v)$ la distanza (numero di archi) da u a v nella rete residua G_f. Si mostri che GENERIC-PREFLOW-PUSH preserva la proprietà che $h[u] < |V|$ implica $h[u] \leq \delta_f(u, t)$ e che $h[u] \geq |V|$ implica $h[u] - |V| \leq \delta_f(u, s)$.

* *27.4-8* Come nell'esercizio precedente, sia $\delta_f(u, v)$ la distanza da u a v nella rete residua G_f. Si mostri come si può modificare l'algoritmo generico del preflusso affinché preservi la proprietà che $h[u] < |V|$ implica $h[u] = \delta_f(u, t)$ e che $h[u] \geq |V|$ implica $h[u] - |V| = \delta_f(u, s)$. Il tempo totale che la realizzazione proposta impiega per preservare questa proprietà deve essere $O(VE)$.

27.4-9 Si mostri che il numero di invii non saturanti eseguito da GENERIC-PREFLOW-PUSH su di una rete di flusso $G = (V, E)$ è al massimo $4|V|^2 - |E|$, per $|V| \geq 4$.

* 27.5 Algoritmo lift-to-front

Il metodo dei preflussi ci consente di applicare le operazioni di base in un qualunque ordine. Scegliendo con attenzione l'ordine e gestendo la struttura di dati della rete in modo efficiente, possiamo risolvere il problema di flusso massimo in tempo minore del limite di $O(V^2E)$ fornito dal corollario 27.25. Esamineremo ora l'algoritmo lift-to-front, un algoritmo di preflusso il cui tempo di esecuzione è $O(V^3)$ che è asintoticamente migliore di $O(V^2E)$.

L'algoritmo lift-to-front mantiene una lista contenente tutti i vertici del la rete. L'algoritmo scandisce la lista cominciando dalla testa, selezionando ripetutamente un vertice traboccante u e quindi "scaricandolo", cioè, eseguendo operazioni di invio e di innalzamento finché u non ha più un eccesso positivo. Quando un vertice viene innalzato, esso viene spostato in testa alla lista (da qui il nome inglese dell'algoritmo, "lift-to-front") e l'algoritmo ricomincia a scorrere la lista.

La correttezza e l'analisi dell'algoritmo lift-to-front dipendono dalla nozione di archi "ammissibili": sono quegli archi nella rete residua attraverso i quali può essere inviato del flusso. Dopo aver dimostrato alcune proprietà riguardanti la rete degli archi ammissibili, esamineremo l'operazione di scaricamento e quindi presenteremo ed analizzeremo l'algoritmo lift-to-front.

Archi e reti ammissibili

Se $G = (V, E)$ è una rete di flusso con sorgente s e pozzo t, f è un preflusso in G e h è una funzione altezza, diciamo che (u, v) è un ***arco ammissibile*** se $c_f(u, v) > 0$ e $h(u) = h(v) + 1$. Altrimenti (u, v) è ***non ammissibile***. La ***rete ammissibile*** è $G_{f,h} = (V, E_{f,h})$, dove $E_{f,h}$ è l'insieme degli archi ammissibili.

La rete ammissibile consiste di tutti gli archi attraverso i quali può essere inviato del flusso. Il lemma seguente mostra che questa rete è un grafo orientato aciclico (dag).

Lemma 27.26 (*La rete ammissibile è aciclica*)
Se $G = (V, E)$ è una rete di flusso, f è un preflusso in G e h è una funzione altezza su G, allora la rete ammissibile $G_{f,h} = (V, E_{f,h})$ è aciclica.

Dimostrazione. La dimostrazione è per assurdo. Si supponga che $G_{f,h}$ contenga un ciclo $p = \langle v_0, v_1, \ldots, v_k \rangle$, dove $v_0 = v_k$ e $k > 0$. Poiché ogni arco in p è ammissibile, abbiamo che $h(v_{i-1}) = h(v_i) + 1$, per $i = 1, 2, \ldots, k$. Sommando lungo il ciclo, otteniamo

$$\begin{aligned} \sum_{i=1}^{k} h(v_{i-1}) &= \sum_{i=1}^{k} (h(v_i) + 1) \\ &= \sum_{i=1}^{k} h(v_i) + k \,. \end{aligned}$$

Poiché ogni vertice del ciclo p compare una volta in ognuna delle sommatorie, ne deriva la contraddizione che $0 = k$. ■

I prossimi due lemmi mostrano come le operazioni di invio e di innalzamento cambino la rete ammissibile.

Lemma 27.27
Sia $G = (V, E)$ una rete di flusso, sia f un preflusso in G e sia h una funzione altezza. Se un vertice u è traboccante e (u, v) è un arco ammissibile, allora PUSH(u, V) può essere applicata. L'operazione non crea nessun arco ammissibile nuovo, ma può far diventare (u, v) non ammissibile.

Dimostrazione. Per la definizione di arco ammissibile, del flusso può essere inviato da u a v; inoltre, poiché u è traboccante, l'operazione PUSH(u, v) può essere applicata. L'unico nuovo arco residuo che può essere creato inviando del flusso da u a v è l'arco (v, u), ma poiché $h(v) = h(u) - 1$, l'arco (v, u) non può diventare ammissibile. Se l'operazione è un invio saturante, allora al termine si ha $c_f(u, v) = 0$ e l'arco (u, v) diventa non ammissibile. ■

Lemma 27.28
Sia $G = (V, E)$ una rete di flusso, sia f un preflusso in G e sia h una funzione altezza. Se un vertice u è traboccante e non vi è alcun arco ammissibile uscente da u, allora l'operazione LIFT(u) può essere applicata. Dopo questa operazione vi è almeno un arco ammissibile uscente da u, ma non vi sono archi ammissibili entranti in u.

Dimostrazione. Se u è traboccante, allora, per il lemma 27.14, si può applicare ad esso o un'operazione di invio o una di innalzamento. Se non vi sono archi ammissibili uscenti da u, allora non si può inviare del flusso da u e quindi si deve poter applicare LIFT(u). Dopo l'operazione di innalzamento, abbiamo che $h[u] = 1 + \min\{h[v] : (u, v) \in E_f\}$; quindi, se v è il vertice che realizza il minimo in questo insieme, l'arco (u, v) diventa ammissibile. Quindi, dopo l'innalzamento vi è almeno un arco ammissibile uscente da u.

Per mostrare che non vi sono archi ammissibili entranti in u dopo l'operazione di innalzamento, si supponga che esista un vertice v tale che (v, u) è ammissibile. Allora dopo l'innalzamento abbiamo $h[v] = h[u] + 1$ e quindi subito prima dell'operazione doveva essere $h[v] > h[u] + 1$. Ma, per il lemma 27.13, non può esistere alcun arco residuo tra vertici le cui altezze differiscano per più di 1. Inoltre, l'innalzamento di un vertice non cambia la rete residua. Quindi, (v, u) non è nella rete residua e a maggior ragione non può essere nella rete ammissibile.

Liste dei vicini

Nell'algoritmo lift-to-front, gli archi sono organizzati in "liste di vicini". Data una rete di flusso $G = (V, E)$, la ***lista dei vicini*** $N[u]$, per un vertice $u \in V$, è una lista concatenata contenente i vicini di u in G; quindi, un vertice v compare nella lista $N[u]$ se $(u, v) \in E$ o $(v, u) \in E$. La lista dei vicini $N[u]$ contiene esattamente quei vertici v per i quali può esistere un arco residuo (u, v). Il primo vertice in $N[u]$ è puntato da $head[N[u]]$, mentre il vertice che segue v in una lista di vicini è puntato da $next\text{-}neighbor[v]$; questo puntatore è NIL se v è l'ultimo vertice della lista.

L'algoritmo lift-to-front scandisce ripetutamente ogni lista di vicini in un ordine arbitrario che è fissato per tutta l'esecuzione dell'algoritmo. Per ogni vertice u, il campo $current[u]$ punta al vertice di $N[u]$ attualmente considerato; all'inizio a $current[u]$ viene assegnato $head[N[u]]$.

Scaricamento di un vertice traboccante

Un vertice traboccante u viene ***scaricato*** inviando tutto il suo flusso in eccesso a vertici vicini attraverso archi ammissibili; ciò si ottiene innalzando u di quanto necessario per rendere ammissibili gli archi uscenti da u. Il seguente pseudocodice realizza queste operazioni.

```
DISCHARGE(u)
while e[u] > 0
    do  v ← current[u]
        if v = NIL
            then  LIFT(u)
                  current[u] ← head[N[u]]
        else if c_f(u, v) > 0 e h[u] = h[v] + 1
            then  PUSH(u, V)
        else current[u] ← next-neighbor[v]
```

Figura 27.10 *Scaricamento di un vertice. Per inviare tutto il flusso in eccesso dal vertice y sono necessarie* 15 *iterazioni del ciclo* **while** *di* DISCHARGE. *Nella figura sono mostrati solo i vicini di y e gli archi entranti in y o uscenti da esso. In ogni parte della figura, il numero all'interno di ogni vertice è il suo eccesso all'inizio della prima iterazione mostrata in quella parte e ogni vertice è mostrato alla sua altezza che rimane invariata per quella parte. A destra è mostrata la lista dei vicini N[y] all'inizio di ogni iterazione, con il numero di iterazione in alto. Il vicino in neretto è current[y]. (a) Inizialmente ci sono* 19 *unità in eccesso che devono essere inviate da y e current[y] = s. Le iterazioni* 1, 2 *e* 3 *fanno solo avanzare current[y] poiché non vi sono archi ammissibili uscenti da y. Nell'iterazione* 4, *c current[y] =* NIL *(infatti nessun vicino è in neretto nella lista) e quindi y viene innalzato e current[y] viene ripristinato con l'elemento in testa alla lista dei vicini. (b) Dopo l'innalzamento, il vertice y ha altezza* 1. *Nelle iterazioni* 5 *e* 6, *gli archi (y, s) e (y, x) risultano non ammissibili, ma* 8 *unità di flusso possono essere inviate da y a z all'iterazione* 7. *A causa dell'invio, current[y] non viene spostato in questa iterazione. (c) Poiché l'invio di flusso all'iterazione* 7 *ha saturato l'arco (y, z) esso risulta non ammissibile nell'iterazione* 8. *Nell'iterazione* 9 *abbiamo che current[y] =* NIL, *quindi il vertice y viene nuovamente innalzato, e current[y] viene ripristinato. (d) Nell'iterazione* 10, *(y, s) è non ammissibile, ma* 5 *unità di flusso in eccesso sono inviate da y a x nell'iterazione* 11. *(e) Poiché current[y] non era stato spostato nell'iterazione* 11, *l'iterazione* 12 *trova che (y, v) non è ammissibile; l'iterazione* 13 *trova (y, z) non ammissibile e l'iterazione* 14 *innalza il vertice y e ripristina current[y]. (f) L'iterazione* 15 *invia* 6 *unità di flusso in eccesso da y a s. (g) Il vertice y non ha più flusso in eccesso, quindi* DISCHARGE *termina. In questo esempio,* DISCHARGE *parte e termina con il puntatore current[y] posizionato sulla testa della lista dei vicini, ma in generale ciò non è necessariamente vero.*

La figura 27.10 mostra diverse iterazioni del ciclo **while** alle linee 1-8 il quale viene eseguito finché il vertice u ha un eccesso positivo. Ogni iterazione esegue esattamente una tra tre possibili azioni, scelta sulla base del vertice corrente v nella lista dei vicini $N[u]$.

Figura 27.10 *(continua)*

1. Se v è NIL, allora è stata raggiunta la fine della lista $N[u]$. In questo caso la linea 4 innalza il vertice u, mentre la linea 5 stabilisce che il prossimo vicino di u da considerare è il primo elemento di $N[u]$, assegnandolo alla variabile *current*[u]. (Il lemma 27.29 che segue mostra che in questa situazione è applicabile l'operazione di innalzamento.)
2. Se v è diverso da NIL e (u, v) è un arco ammissibile (questo viene determinato dalla verifica alla linea 6), allora la linea 7 invia parte dell'eccesso di u (o tutto l'eccesso, se possibile) al vertice v.
3. Se v è diverso da NIL ma (u, v) non è ammissibile, allora la linea 8 ***avanza*** *current*[u] di una posizione nella lista dei vicini $N[u]$.

Si osservi che se DISCHARGE viene invocata su un vertice traboccante u, allora l'ultima azione effettuata da DISCHARGE deve essere un invio da u. Infatti, la procedura termina solo se $e[u]$ diventa nullo e né l'operazione di innalzamento né l'avanzamento del puntatore *current*[u] modificano il valore di $e[u]$.

Bisogna essere sicuri che quando PUSH o LIFT vengono chiamate da DISCHARGE, le corrispondenti operazioni siano applicabili. Il seguente lemma garantisce questo fatto.

Lemma 27.29

Se DISCHARGE invoca PUSH(u, v) alla linea 7, allora un'operazione di invio è applicabile a (u, v). Inoltre, se DISCHARGE chiama LIFT(u) alla linea 4, allora un'operazione di innalzamento può essere applicata ad u.

Dimostrazione. Le verifiche alle linee 1 e 6 assicurano che un'operazione di invio venga eseguita solo se è applicabile e ciò dimostra la prima parte del lemma.

Per dimostrare la seconda parte, sulla base della verifica alla linea 1 e del lemma 27.28, dobbiamo solo mostrare che tutti gli archi che escono da u sono non ammissibili. Si osservi che, durante le ripetute chiamate alla procedura DISCHARGE(u), il puntatore *current*[u] scandisce più volte la lista $N[u]$. Ogni "passata" comincia dalla testa di $N[u]$ e termina con *current*[u] = NIL, dopodiché u viene innalzato e comincia una nuova passata. Affinché il puntatore *current*[u] possa avanzare oltre un vertice $v \in N[u]$ durante una passata, l'arco (u, v) deve essere riconosciuto come non ammissibile dalla verifica alla linea 6; di conseguenza, quando una passata termina, ogni arco uscente da u deve essere stato riconosciuto come non ammissibile in un qualche istante durante la passata. L'osservazione chiave è che, al termine della passata, ogni arco uscente da u è ancora non ammissibile. Infatti, per il lemma 27.27, gli invii non possono creare alcun arco ammissibile, tanto meno un arco che esca da u. Quindi, un qualunque arco ammissibile dev'essere stato creato da un'operazione di innalzamento. Ma il vertice u non viene innalzato durante la passata e, per il lemma 27.28, ogni altro vertice v innalzato durante la passata non ha archi ammissibili entranti. Quindi, al termine della passata, tutti gli archi uscenti da u rimangono non ammissibili ed il lemma è dimostrato. ■

Algoritmo lift-to-front

Nell'algoritmo lift-to-front, manterremo una lista concatenata L contenente tutti i vertici in $V - \{s, t\}$. Una proprietà fondamentale è che i vertici in L sono ordinati topologicamente rispetto alla rete ammissibile (si ricordi, dal lemma 27.26, che la rete ammissibile è un dag).

Lo pseudocodice per l'algoritmo lift-to-front assume che la lista dei vicini $N[u]$ sia già stata creata per ogni vertice u. Esso assume anche che *next*[u] punti al vertice che segue u nella lista L e che, come al solito, *next*[u] = NIL se u è l'ultimo vertice della lista.

```
LIFT-TO-FRONT(G, s, t)
1   INITIALIZE-PREFLOW(G, s)
2   L ← V[G] − {s, t}, in qualunque ordine
3   for ogni vertice u ∈ V[G] − {s, t}
4           do current[u] ← head[N[u]]
5   u ← head[L]
6   while u ≠ NIL
7           do old-height ← h[u]
8                   DISCHARGE(u)
9                   if h[u] > old-height
10                          then sposta u in testa alla lista L
11                  u ← next[u]
```

L'algoritmo lift-to-front funziona nel modo seguente. La linea 1 inizializza il preflusso e le altezze con gli stessi valori usati dall'algoritmo generico di preflusso. La linea 2 inizializza la lista L con tutti i vertici potenzialmente traboccanti, in un qualunque ordine. Le linee 3-4 inizializzano il puntatore *current* di ogni vertice u con il primo vertice nella lista di vicini di u. Come mostrato nella figura 27.11, il ciclo **while** alle linee 6-11 avanza lungo la lista L, scaricando i vertici uno alla volta. Infatti, la linea 5 fa partire tale ciclo dal primo vertice della lista e ad ogni iterazione del ciclo un vertice u viene scaricato alla linea 8. Se u era stato innalzato dalla procedura DISCHARGE, la linea 10 lo sposta in testa alla lista L. Questa decisione viene presa memorizzando l'altezza di u nella variabile *old-height* prima dell'operazione di scaricamento (linea 7) e confrontando questo valore con l'altezza di u dopo l'operazione (linea 9). La linea 11 fa in modo che la prossima iterazione del ciclo **while** usi il vertice che segue u nella lista L. Se u era stato spostato in testa alla lista, il vertice usato alla prossima iterazione è quello che segue u nella sua nuova posizione nella lista.

Per mostrare che LIFT-TO-FRONT calcola un flusso massimo, dimostreremo che tale algoritmo è una realizzazione dell'algoritmo generico del preflusso. Per prima cosa si osservi che esso effettua operazioni di invio e di innalzamento solo quando esse sono applicabili, ciò dal lemma 27.29. Rimane da mostrare che quando LIFT-TO-FRONT termina, nessuna operazione di base è più applicabile. Si osservi che se u raggiunge la fine di L, ogni vertice in L dev'essere stato scaricato senza causare un innalzamento.

Il lemma 27.30, che dimostreremo tra poco, garantisce che la lista L contiene ad ogni istante un ordinamento topologico della rete ammissibile. Quindi un'operazione di invio fa muovere flusso in eccesso in avanti lungo la lista (oppure verso s o t). Così, se il puntatore u raggiunge la fine della lista, l'eccesso di ogni vertice dev'essere 0; di conseguenza, nessuna operazione di base è più applicabile.

Lemma 27.30

Se si esegue LIFT-TO-FRONT su una rete di flusso $G = (V, E)$ con sorgente s e pozzo t, allora ogni iterazione del ciclo **while** alle linee 1-6 preserva la proprietà invariante che la lista L è un ordinamento topologico dei vertici nella rete ammissibile $G_{f,h} = (V, E_{f,h})$.

Dimostrazione. Immediatamente dopo che INITIALIZE-PREFLOW è stata eseguita, $h[s] = |V|$ e $h[v] = 0$, per ogni $v \in V - \{s\}$. Poiché $|V| \geq 2$ (perché V contiene almeno s e t), nessun arco può essere ammissibile. Quindi, $E_{f,h} = \emptyset$ ed un qualunque ordinamento di $V - \{s, t\}$ è un ordinamento topologico di $G_{f,h}$.

Mostriamo ora che la proprità invariante viene mantenuta da ogni iterazione del ciclo **while**. La rete ammissibile viene cambiata solo da operazioni di invio e di innalzamento. Per il lemma 27.27, le operazioni di invio possono solo rendere degli archi non ammissibili e quindi nuovi archi ammissibili possono essere creati solo da operazioni di innalzamento. Dopo che

(continua a pagina seguente)

Figura 27.11 *L'effetto di* LIFT-TO-FRONT. (a) *Una rete di flusso subito prima della prima operazione del ciclo* **while**. *All'inizio vengono inviate dalla sorgente s 26 unità di flusso. A destra è mostrata la lista iniziale L = ⟨x, y, z⟩ dove all'inizio u = x. Sotto ogni vertice nella lista L è indicata la lista dei vicini, con il vicino corrente in neretto. Il vertice x viene scaricato: esso viene innalzato all'altezza 1, poi 5 unità di flusso vengono inviate a y e le rimanenti 7 unità in eccesso vengono inviate al pozzo t. Poiché x è innalzato, esso viene spostato in testa ad L che in questo caso non cambia struttura.* (b) *Dopo x, il prossimo vertice in L che viene scaricato è y. La figura 27.10 mostra in modo dettagliato l'effetto dell'operazione di scaricamento di y in questa situazione. Poiché y è innalzato, esso viene spostato in testa alla lista.* (c) *Il vertice x ora segue y in L e quindi esso dev'essere nuovamente scaricato, inviando tutte le 5 unità di flusso in eccesso a t. Poiché il vertice x non viene innalzato da questa operazione di scaricamento, esso rimane al suo posto nella lista L.* (d) *Poiché il vertice z segue x in L, esso viene scaricato. Esso viene innalzato all'altezza 1 e tutte le 8 unità di flusso in eccesso vengono inviate a t. Poiché z è stato innalzato, viene spostato in testa a L.* (e) *Il vertice y ora segue z in L e quindi viene scaricato: ma poiché y non ha eccesso, esso rimane al suo posto in L. Quindi viene scaricato il vertice x, ma poiché anch'esso non ha eccesso, esso viene mantenuto al proprio posto in L. A questo punto* LIFT-TO-FRONT *ha raggiunto la fine della lista L e termina. Non vi sono più vertici traboccanti ed il preflusso è un flusso massimo.*

un vertice è stato innalzato, il lemma 27.28 garantisce che non vi siano archi ammissibili entranti in u, mentre ci possono essere archi ammissibili uscenti da u. Quindi, spostando u in testa ad L, l'algoritmo garantisce che qualunque arco ammissibile uscente da u soddisfi l'ordinamento topologico. ■

Figura 27.11 (*continua*)

Analisi

Mostreremo ora che la procedura LIFT-TO-FRONT ha un tempo di esecuzione $O(V^3)$ su una qualunque rete di flusso $G = (V, E)$. Poiché l'algoritmo è una realizzazione dell'algoritmo generico del preflusso, sfrutteremo il corollario 27.21 che fornisce un limite superiore di $O(V)$ al numero di operazioni di innalzamento eseguite per ogni vertice ed un limite di $O(V^2)$ al numero totale di innalzamenti. Inoltre, l'Esercizio 27.4-2 fornisce un limite superiore di $O(VE)$ al tempo totale impiegato per eseguire le operazioni di innalzamento ed il lemma 27.22 dà un limite di $O(VE)$ al numero totale di operazioni di invio saturante.

> ***Teorema 27.31***
>
> Il tempo di esecuzione di LIFT-TO-FRONT su una qualunque rete di flusso $G = (V, E)$ è $O(V^3)$.

Dimostrazione. Si definisca "fase" dell'algoritmo lift-to-front il tempo che trascorre tra due operazioni di innalzamento consecutive. Chiaramente vi sono $O(V^2)$ fasi, poiché vi sono $O(V^2)$ operazioni di innalzamento. Ogni fase consiste al massimo di $|V|$ chiamate a DISCHARGE per il seguente motivo. Se DISCHARGE non esegue un'operazione di innalzamento, la prossima chiamata di DISCHARGE farà riferimento ad un vertice che si trova più avanti nella lista L e la lunghezza di L è minore di $|V|$; se invece DISCHARGE effettua un innalzamento, allora la prossima chiamata di DISCHARGE apparterrà ad una fase diversa. Poiché ogni fase contiene al massimo $|V|$ chiamate a DISCHARGE e vi sono $O(V^2)$ fasi, il numero di volte che DISCHARGE viene invocata alla linea 8 di LIFT-TO-FRONT è $O(V^3)$. Quindi, il lavoro totale effettuato dal ciclo **while** in LIFT-TO-FRONT, escludendo il lavoro fatto all'interno di DISCHARGE, è al massimo $O(V^3)$.

Dobbiamo ora trovare un limite superiore al lavoro effettuato all'interno di DISCHARGE durante l'esecuzione dell'algoritmo. Ogni iterazione del ciclo **while** all'interno di DISCHARGE

esegue una tra tre possibili azioni. Analizzeremo ora la quantità totale di lavoro necessaria per eseguire ognuna di queste azioni.

Cominciamo con le operazioni di innalzamento (linee 4-5). L'Esercizio 27.4-2 fornisce un limite superiore di $O(VE)$ per il tempo necessario ad eseguire gli $O(V^2)$ innalzamenti.

Supponiamo ora che l'azione aggiorni il puntatore *current*[u] alla linea 8. Questa azione avviene O(grado(u)) volte ogni volta che un vertice u viene innalzato, quindi $O(V \cdot$ grado(u)) volte per vertice. Di conseguenza, il lavoro totale necessario per far avanzare i puntatori nelle liste dei vicini di tutti i vertici è $O(VE)$ per il lemma delle stretta di mano (Esercizio 5.4-1).

Il terzo tipo di azione effettuato da DISCHARGE è un'operazione di invio (linea 7). Sappiamo già che il numero totale di operazioni di invio saturante è $O(VE)$. Si osservi inoltre che se viene eseguito un invio non saturante, DISCHARGE ritorna immediatamente, perché un tale invio riduce l'eccesso a 0. Quindi, ci può essere al massimo un invio non saturante per ogni chiamata di DISCHARGE: come abbiamo già osservato, la procedura DISCHARGE viene invocata $O(V^3)$ volte, così il tempo totale impiegato per effettuare degli invii non saturanti è $O(V^3)$.

Il tempo di esecuzione di LIFT-TO-FRONT è quindi $O(V^3 + VE)$ che è uguale a $O(V^3)$. ■

Esercizi

27.5-1 Si illustri l'esecuzione di LIFT-TO-FRONT sulla rete di flusso della figura 27.1(a) procedendo in modo simile alla figura 27.11. Si assuma che l'ordinamento iniziale dei vertici in L sia $\langle v_1, v_2, v_3, v_4 \rangle$ e che le liste dei vicini siano

$$N[v_1] = \langle s, v_2, v_3 \rangle,$$
$$N[v_2] = \langle s, v_1, v_3, v_4 \rangle,$$
$$N[v_3] = \langle v_1, v_2, v_4, t \rangle,$$
$$N[v_4] = \langle v_2, v_3, t \rangle.$$

* *27.5-2* Si vuole realizzare un algoritmo di preflusso in cui si mantiene una coda FIFO di vertici traboccanti. L'algoritmo scarica ripetutamente i vertici dalla testa della coda ed eventuali vertici che non erano traboccanti prima dello scaricamento, ma che lo sono dopo, vengono posti in fondo alla coda. Dopo che il vertice in testa alla coda è stato scaricato, esso viene rimosso; quando la coda è vuota, l'algoritmo termina. Si mostri che questo algoritmo può essere realizzato in modo da trovare un flusso massimo in tempo $O(V^3)$.

27.5-3 Si mostri che l'algoritmo generico funziona ancora se LIFT aggiorna $h[u]$ calcolando semplicemente $h[u] \leftarrow h[u] + 1$. Quale influenza ha questo cambiamento sull'analisi di LIFT-TO-FRONT?

* *27.5-4* Si mostri che se si scarica sempre un vertice traboccante di altezza massima, allora il metodo dei preflussi può essere realizzato in modo da richiedere tempo $O(V^3)$.

(a) (b)

Figura 27.12 *Due griglie per il problema della fuga. I punti di partenza sono neri e gli altri vertici delle griglie sono bianchi.* (a) *Una griglia con una fuga, mostrata dai cammini grigi.* (b) *Una griglia senza fughe.*

Problemi

27-1 Problema della fuga

Una ***griglia*** $n \times n$ è un grafo non orientato che consiste di n righe e n colonne di vertici, come mostrato nella figura 27.12. Il vertice alla riga i-esima e colonna j-esima viene denotato con (i, j). Tutti i vertici nella griglia hanno esattamente quattro vicini, ad eccezione di quelli sul bordo, cioè dei punti (i, j) per cui $i = 1$, $i = n$, $j = 1$ oppure $j = n$.

Dati $m \le n^2$ punti di partenza $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ nella griglia, il ***problema della fuga*** consiste nel determinare se vi siano o meno m cammini con vertici disgiunti dai punti di partenza ad m punti distinti sul bordo. Ad esempio, la griglia della figura 27.12(a) ha una fuga, ma la griglia della figura 27.12(b) non ne ha alcuna.

a. Si consideri una rete di flusso in cui anche i vertici, oltre agli archi, hanno una capacità. Più precisamente, il flusso netto positivo entrante in un qualunque vertice è soggetto ad un vincolo di capacità. Si mostri che il problema di determinare il flusso massimo in una rete con capacità su archi e vertici può essere ridotto ad un ordinario problema di flusso massimo su una rete di flusso di dimensione comparabile.

b. Si descriva un algoritmo efficiente per risolvere il problema della fuga e se ne analizzi il tempo di esecuzione.

27-2 Copertura minima con cammini

Una ***copertura con cammini*** di un grafo orientato $G = (V, E)$ è un insieme P di cammini con vertici disgiunti tale che ogni vertice in V compare in esattamente un cammino in P. I cammini possono cominciare e finire ovunque, e possono avere qualunque lunghezza, compreso 0. Una ***copertura minima con cammini*** di G è una copertura con cammini che contiene il minimo numero possibile di cammini.

a. Si dia un algoritmo efficiente per trovare una copertura minima con cammini di un grafo orientato aciclico $G = (V, E)$. (*Suggerimento*: assumendo che $V = \{1, 2, \ldots, n\}$, si costruisca il grafo $G' = (V', E')$, dove

$$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\} ,$$
$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\}$$

e si esegua un algoritmo di flusso massimo).

b. L'algoritmo proposto funziona anche per grafi orientati che contengono cicli? Si motivi la risposta.

27.3 *Esperimenti con navette spaziali*

Il Professor Spock è un consulente della NASA la quale sta organizzando una serie di voli di navette spaziali e deve decidere quali esperimenti commerciali effettuare e quali strumenti devono essere a bordo di ogni volo. Per ogni volo, la NASA considera un insieme $E = \{E_1, E_2, \ldots, E_m\}$ di esperimenti e lo sponsor commerciale dell'esperimento E_j ha convenuto di pagare alla NASA p_j dollari per i risultati dell'esperimento. Gli esperimenti usano un insieme $I = \{I_1, I_2, \ldots, I_n\}$ di strumenti; ogni esperimento E_j richiede tutti gli strumenti in un sottoinsieme $R_j \subseteq I$. Il costo per portare su una navetta lo strumento I_k è di c_k dollari. Il compito del professore consiste nel trovare un algoritmo efficiente per determinare quali esperimenti effettuare e quali strumenti portare a bordo di un certo volo in modo da massimizzare il ricavato netto, cioè il ricavato totale per gli esperimenti effettuati meno il costo totale per gli strumenti trasportati.

Si consideri la seguente rete G. La rete contiene un vertice sorgente s, i vertici $I_1, I_2, \ldots, I_n$, i vertici $E_1, E_2, \ldots, E_m$ ed un vertice pozzo t. Per $k = 1, 2, \ldots, n$, vi è un arco (s, I_k) di capacità c_k e, per $j = 1, 2, \ldots, m$, vi è un arco (I_k, E_j) di capacità p_j. Inoltre, per $k = 1, 2, \ldots, n$ e $j = 1, 2, \ldots, m$, se $I_k \in R_j$, allora vi è un arco (I_k, E_j) di capacità infinita.

a. Si mostri che se $E_j \in T$ per un taglio (S, T) di G di capacità finita, allora $I_k \in T$ per ogni $I_k \in R_j$.

b. Si mostri come determinare il ricavato netto massimo dalla capacità del taglio minimo di G e dai valori p_j prefissati.

c. Si dia un algoritmo efficiente per determinare quali esperimenti effettuare e quali strumenti trasportare. Si analizzi il tempo di esecuzione dell'algoritmo proposto in termini di m, n, e di $r = \sum_{j=1}^{m} |R_j|$.

27.4 *Aggiornamento del flusso massimo*

Sia $G = (V, E)$ una rete di flusso con sorgente s, pozzo t e capacità intere. Si supponga di aver calcolato un flusso massimo in G.

a. Si supponga che la capacità di un singolo arco $(u, v) \in E$ venga incrementata di 1. Si fornisca un algoritmo per aggiornare il flusso massimo in tempo $O(V + E)$.

b. Si supponga che la capacità di un singolo arco $(u, v) \in E$ venga decrementata di 1. Si fornisca un algoritmo per aggiornare il flusso massimo in tempo $O(V + E)$.

27-5 *Calcolo incrementale del flusso massimo*

Sia $G = (V, E)$ una rete di flusso con sorgente s, pozzo t ed una capacità intera $c(u, v)$ per ogni arco $(u, v) \in E$. Sia $C = \max_{(u,v) \in E} c(u, v)$.

a. Si deduca che un taglio minimo di G ha capacità al più $C|E|$.

b. Per un dato numero K, si mostri che un cammino aumentante di capacità almeno K, se esiste, può essere trovato in tempo $O(E)$.

La seguente variante di FORD-FULKERSON-METHOD può essere usata per calcolare un flusso massimo in G.

```
MAX-FLOW-BY-SCALING(G, s, t)
1  C ← max_{(u,v)∈E} c(u, v)
2  inizializza il flusso f a 0
3  K ← 2^⌊lg C⌋
4  while K > 1
5      do while esiste un cammino aumentante p di capacità almeno K
6          do aumenta il flusso f lungo p
7          K ← K/2
8  return f
```

c. Si dimostri che MAX-FLOW-BY-SCALING calcola un flusso massimo.

d. Si mostri che la capacità di un taglio minimo della rete residua G_f è al più $2K|E|$ ogni volta che la linea 4 viene eseguita.

e. Si deduca che il ciclo **while** più interno, alle linee 5-6, viene eseguito $O(E)$ volte per ogni valore di K.

f. Si concluda che MAX-FLOW-BY-SCALING può essere realizzato in modo da richiedere tempo $O(E^2 \lg C)$.

27-6 *Flusso massimo con capacità superiori ed inferiori*

Si supponga che ogni arco (u, v) in una rete di flusso $G = (V, E)$ abbia non solo una capacità superiore $c(u, v)$ per il flusso netto da u a v, ma anche una capacità inferiore $b(u, v)$; quindi, un qualunque flusso f sulla rete deve soddisfare $b(u, v) \le f(u, v) \le c(u, v)$. Per una rete di questo tipo, può anche succedere che non esista alcun flusso.

a. Si dimostri che se f è un flusso su G, allora $|f| \le c(S, T) - b(T, S)$ per un qualunque taglio (S, T) di G.

b. Si dimostri che il valore di un flusso massimo nella rete, se esiste, è il valore minimo di $c(S, T) - b(T, S)$ calcolato su tutti i tagli (S, T) della rete.

Sia $G = (V, E)$ una rete di flusso con funzioni di capacità superiore ed inferiore c e b e siano s e t la sorgente ed il pozzo di G. Si costruisca la rete di flusso ordinaria $G' = (V', E')$ con funzione di capacità superiore c', sorgente s' e pozzo t' nel modo seguente:

$$V' = V \cup \{s', t'\} ,$$
$$E' = E \cup \{(s', v) : v \in V\} \cup \{(u, t') : u \in V\} \cup \{(s, t), (t, s)\} .$$

Le capacità vengono assegnate agli archi nel modo seguente: per ogni arco $(u, v) \in E$, si pone $c'(u, v) = c(u, v) - b(u, v)$.

Per ogni vertice $u \in V$, si pone $c'(s', u) = b(V, u)$ e $c'(u, t') = b(u, V)$. Inoltre, poniamo $c'(s, t) = c'(t, s) = \infty$.

c. Si dimostri che esiste un flusso in G se e solo se esiste un flusso massimo in G' in cui tutti gli archi verso il pozzo t' sono saturi.

d. Si dia un algoritmo che trovi un flusso massimo in una rete con capacità inferiori e superiori, o che determini che non esiste alcun flusso ammissibile. Si analizzi il tempo di esecuzione dell'algoritmo proposto.

Note al capitolo

Even [65], Lawler [132], Papadimitriou e Steiglitz [154] e Tarjan [188] sono buoni riferimenti per reti di flusso ed algoritmi relativi. Goldberg, Tardos e Tarjan [83] forniscono una bella rassegna di algoritmi per problemi di reti di flusso.

Il metodo di Ford-Fulkerson è dovuto a Ford e Fulkerson [71] i quali hanno formalizzato molti problemi nell'area delle reti di flusso, compresi i problemi di flusso massimo e di abbinamento in grafi bipartiti. Molte tra le prime realizzazioni del metodo di Ford-Fulkerson trovavano i cammini aumentanti usando una visita in ampiezza; Edmonds e Karp [63] hanno dimostrato che questa strategia produce un algoritmo polinomiale in tempo. Karzanov [119] ha sviluppato l'idea dei preflussi; il metodo dei preflussi è dovuto a Goldberg [82]. Il più veloce algoritmo di preflusso proposto sinora è dovuto a Goldberg e Tarjan [85] che ottengono un tempo d'esecuzione $O(VE \lg(V^2/E))$. Il migliore algoritmo proposto sinora per l'abbinamento bipartito massimo, scoperto da Hopcroft e Karp [101], richiede tempo $O(\sqrt{V}E)$.

VII Complementi ed estensioni

Introduzione

Questa parte affronta una serie di algoritmi che estendono e completano il materiale iniziale di questo libro. Alcuni capitoli introducono nuovi modelli di calcolo come i circuiti combinatori o i calcolatori paralleli. Altri coprono argomenti specialistici come la geometria computazionale o la teoria dei numeri. Gli ultimi due capitoli discutono alcune note limitazioni al progetto di algoritmi efficienti e introducono tecniche per affrontare queste limitazioni.

Il Capitolo 28 presenta il primo modello di calcolo parallelo: le reti di confrontatori. In parole povere, una rete di confrontatori è un algoritmo che consente di eseguire contemporaneamente molti confronti. Questo capitolo mostra come costruire una rete di confrontatori che possa ordinare n numeri in tempo $O(\lg^2 n)$.

Il Capitolo 29 introduce un altro modello di calcolo parallelo: i circuiti combinatori. Questo capitolo mostra che due numeri di n cifre binarie possono essere addizionati in tempo $O(\lg n)$ usando un circuito combinatorio chiamato addizionatore con previsione del riporto. Inoltre vedremo come moltiplicare due numeri di n cifre binarie in tempo $O(\lg n)$.

Il Capitolo 30 introduce un modello generale di calcolo parallelo chiamato PRAM. Il capitolo presenta le tecniche parallele di base, comprese quelle del salto dei puntatori, del calcolo dei prefissi e la tecnica del ciclo di Eulero. Molte tecniche sono descritte utilizzando semplici strutture dati, incluse le liste e gli alberi. Il capitolo discute anche aspetti generali del calcolo parallelo, inclusi l'efficienza e l'accesso concorrente alla memoria condivisa. Viene dimostrato il teorema di Brent che mostra come un calcolatore parallelo possa simulare efficientemente un circuito combinatorio. Il capitolo si conclude con un efficiente algoritmo randomizzato per il calcolo dei suffissi e un algoritmo deterministico notevolmente efficiente per rompere la simmetria in una lista.

Il Capitolo 31 studia algoritmi efficienti per operare sulle matrici. Comincia con l'algoritmo di Strassen che può moltiplicare due matrici $n \times n$ in tempo $O(n^{2.81})$. Presenta inoltre due metodi generali – decomposizione LU e decomposizione LUP – per risolvere equazioni lineari con il metodo di eliminazione di Gauss in tempo $O(n^3)$. Mostra anche che l'algoritmo di Strassen può essere usato per risolvere più efficientemente i sistemi lineari e che l'inversione e la moltiplicazione di matrici possono essere eseguite, asintoticamente, in modo ugualmente efficiente.

Il Capitolo 32 studia le operazioni sui polinomi e mostra che una ben nota tecnica di gestione dei segnali – la Trasformata Veloce di Fourier (detta anche FFT, dall'inglese Fast Fourier Transform) – può essere usata per moltiplicare due polinomi di grado n in tempo $O(n\lg n)$. Inoltre, studia realizzazioni efficienti dell'FFT, compreso un circuito parallelo.

Il Capitolo 33 presenta algoritmi che si basano sulla teoria dei numeri. Dopo una rassegna sulla teoria elementare dei numeri, presenta l'algoritmo di Euclide per calcolare il massimo comun divisore. Nel seguito vengono mostrati gli algoritmi per risolvere le equazioni lineari modulari e per elevare un numero a potenza modulo un altro numero. È quindi presentata una applicazione interessante di algoritmi basati sulla teoria dei numeri: la crittografia a chiave pubblica RSA. Questo sistema non solo può essere usato per scrivere messaggi in codice in modo che nessuno possa leggerli, ma anche per fornire firme numeriche. Il capitolo presenta quindi la verifica randomizzata di primalità di Miller-Rabin che può essere usata per trovare in modo efficiente numeri primi grandi – richiesta essenziale del sistema RSA. Infine, il capitolo affronta l'euristica "rho" di Pollard per scomporre numeri interi in fattori primi e discute lo stato dell'arte della scomposizione di interi in fattori.

Il Capitolo 34 studia il problema della corrispondenza tra stringhe di caratteri (in inglese "string matching") che consiste nel trovare tutte le occorrenze di una stringa all'interno di un'altra stringa e che si presenta frequentemente nei programmi di scrittura di testi. Viene prima presentato un elegante approccio di Rabin e Karp. Quindi, dopo aver esaminato una soluzione efficiente basata su automi a stati finiti, il capitolo presenta l'algoritmo di Knuth-Morris-Pratt che raggiunge l'efficienza ottima in modo astuto attraverso una pre-elaborazione della stringa di input. Il capitolo si chiude con la presentazione delle euristiche dovute a Boyer e Moore.

La geometria computazionale è l'argomento del Capitolo 35. Dopo aver discusso le primitive di base della geometria computazionale, il capitolo mostra come un metodo di "rastrellamento" possa determinare in modo efficiente se un insieme di segmenti presenta qualche intersezione. Due interessanti algoritmi per trovare un inviluppo convesso di un insieme di punti – algoritmo di Graham e algoritmo di Jarvis - mostrano la potenza dei metodi di rastrellamento. Il capitolo si chiude con un algoritmo efficiente per trovare la coppia più vicina in un dato insieme di punti nel piano.

Il Capitolo 36 considera i problemi NP-completi. Molti interessanti problemi computazionali sono NP-completi, ma non si conosce alcun algoritmo che risolva qualcuno di questi problemi in tempo polinomiale. Questo capitolo presenta le tecniche per determinare quando un problema è NP-completo. Alcuni problemi classici sono stati dimostrati essere NP-completi: determinare se un grafo ha un ciclo di Hamilton, determinare se una formula booleana è soddisfattibile e determinare se un dato insieme di numeri ha un sottoinsieme i cui valori danno come somma un dato valore. Il capitolo inoltre dimostra che il famoso problema del commesso viaggiatore è NP-completo.

Il Capitolo 37 mostra come algoritmi approssimati possano essere usati per trovare efficientemente soluzioni approssimate di problemi NP-completi. Per qualche problema NP-completo, soluzioni approssimate che siano vicine a quelle ottime sono abbastanza facili da produrre, ma per altri problemi anche algoritmi che diano soluzioni approssimate sempre migliori funzionano sempre peggio al crescere della dimensione del problema. Quindi, vi sono alcuni problemi per cui si può impiegare una quantità crescente di tempo di calcolo in cambio di soluzioni approssimate sempre migliori. Questo capitolo mostra queste possibilità attraverso il problema della copertura di vertici, il problema del commesso viaggiatore, il problema della copertura di insieme e il problema della somma di sottoinsieme.

Reti di confrontatori

28

Nella Parte II sono stati esaminati gli algoritmi di ordinamento per calcolatori seriali (macchine RAM) che consentono l'esecuzione di una sola operazione alla volta. In questo capitolo, si affronteranno algoritmi di ordinamento basati sul modello di calcolo a rete di confrontatori in cui molte operazioni di confronto possono essere eseguite contemporaneamente.

Le reti di confrontatori differiscono dalla macchina RAM in due importanti aspetti. In primo luogo, essi possono eseguire solo confronti. Quindi un algoritmo come il counting sort (si veda il paragrafo 9.2) non può essere realizzato su una rete di confrontatori. In secondo luogo, diversamente dal modello RAM in cui le operazioni sono seriali - cioè una dopo l'altra - le operazioni in una rete di confrontatori possono essere eseguite simultaneamente, o "in parallelo". Come si vedrà, questa caratteristica consente la costruzione di reti di confrontatori che ordinano n valori in tempo sub-lineare.

Si comincia al paragrafo 28.1 con la definizione delle reti di confrontatori e delle reti di ordinamento. Viene anche data una definizione naturale del "tempo di esecuzione" di una rete di confrontatori in termini della profondità della rete. Il paragrafo 28.2 dimostra il "principio zero-uno" che facilita fortemente il compito di analizzare la correttezza delle reti di ordinamento. Sarà progettata una rete di ordinamento efficiente che è, essenzialmente, una versione parallela dell'algoritmo merge-sort del paragrafo 1.3.1. La costruzione prevede tre passi. Il paragrafo 28.3 presenta il progetto di un ordinatore "bitonico" che è il blocco di base della costruzione. Si modifica leggermente l'ordinatore bitonico nel paragrafo 28.4 per produrre una rete di fusione che può fondere due sequenze ordinate in una sequenza ordinata. Infine, nel paragrafo 28.5, si assemblano queste reti di fusione in una rete di ordinamento che può ordinare n valori in tempo $O(\lg^2 n)$.

28.1 Reti di confrontatori

Le reti di ordinamento sono reti di confrontatori che ordinano sempre i loro input; cominceremo quindi con la descrizione delle reti di confrontatori e delle loro caratteristiche. Una rete di confrontatori è costituita esclusivamente di fili elettrici e confrontatori. Un ***confrontatore***, mostrato nella figura 28.1(a), è un dispositivo con due input, x e y, e due output x' e y', che esegue la funzione seguente:

$$x' = \min(x, y),$$
$$y' = \max(x, y).$$

Figura 28.1 (a) *Un confrontatore con input x e y e output x' e y'.* (b) *Lo stesso confrontatore disegnato come una linea verticale. Sono mostrati gli input x = 7, y = 3 e gli output x' = 3, y' = 7.*

Poiché la rappresentazione grafica di un confrontatore nella figura 28.1(a) è troppo grande, si adotterà la convenzione di disegnare i confrontatori con una semplice linea verticale, come mostrato nella figura 28.1(b). Gli input appaiono sulla sinistra e gli output sulla destra. il valore di input più piccolo appare sull'output in alto e il valore di input più grande appare sull'output in basso. Si può pertanto pensare ad un confrontatore come a un ordinatore dei suoi due input.

Si assumerà che ogni confrontatore operi in tempo $O(1)$. In altre parole, si assume che il tempo tra la comparsa dei valori x e y di input e la produzione dei valori x' e y' di output sia costante.

Un ***filo elettrico*** (che chiameremo anche semplicemente ***filo***) trasmette un valore da un posto all'altro. I fili possono connettere l'output di un confrontatore all'input di un altro; in caso contrario essi sono fili di input o di output della rete. In questo capitolo si assumerà che una rete di confrontatori contenga ***n fili di input*** $a_1, a_2, \ldots, a_n$, attraverso i quali i valori da ordinare entrano nella rete, e ***n fili di output*** $b_1, b_2, \ldots, b_n$ che producono il risultato calcolato dalla rete. Inoltre si parlerà di ***sequenza di input*** $\langle a_1, a_2, \ldots, a_n \rangle$ e di ***sequenza di output*** $\langle b_1, b_2, \ldots, b_n \rangle$ per riferirsi ai valori sui fili di input e di output. Cioè, si userà lo stesso nome sia per il filo che per il valore trasmesso. Il significato sarà sempre chiaro dal contesto.

La figura 28.2 mostra una ***rete di confrontatori*** che è un insieme di confrontatori interconnessi tramite fili. Si rappresenta una rete di confrontatori su n input come un insieme di ***n linee*** orizzontali con i confrontatori disposti verticalmente. Si noti che una linea *non* rappresenta un singolo filo, ma piuttosto una serie di fili distinti che connettono i vari confrontatori. La linea in alto della figura 28.2, per esempio, rappresenta tre fili: il filo di input a_1, che si connette a un input del confrontatore A, il filo che connette l'output in alto del confrontatore A ad un input del confrontatore C e il filo di output b_1, che esce dall'output del confrontatore C. Ciascun input di un confrontatore è connesso a un filo che è o uno degli n fili di input $a_1, a_2, \ldots, a_n$ della rete oppure è connesso all'output di un altro confrontatore. Analogamente, ciascun output di un confrontatore è connesso a un filo che è o uno degli n fili di output $b_1, b_2, \ldots, b_n$ della rete oppure è connesso all'input di un altro confrontatore. Il requisito principale dell'interconnessione dei confrontatori è che il grafo di interconnessione deve essere aciclico: se si segue un cammino dall'output di un dato confrontatore all'input di un altro, all'output e poi all'input e così via, il cammino seguito non deve mai tornare indietro formando un ciclo su se stesso né passare attraverso uno stesso confrontatore due volte. Pertanto, come nella figura 28.2, si può disegnare una rete di confrontatori con gli input della rete sulla sinistra e gli output sulla destra; i dati si muovono attraverso la rete da sinistra a destra.

Ogni confrontatore produce i suoi valori di output solo quando entrambi i suoi valori di input sono disponibili. Nella figura 28.2(a) si supponga, per esempio, che la sequenza $\langle 9, 5, 2, 6 \rangle$ appaia sui fili di input al tempo 0. Allora al tempo 0 solo i confrontatori A e B hanno

Figura 28.2 (a) *Una rete di confrontatori con 4 input e 4 output che è in realtà una rete di ordinamento. Al tempo 0 i valori in input mostrati appaiono sui quattro fili di input.* (b) *Al tempo 1, i valori mostrati appaiono sugli output dei confrontatori A e B che sono a profondità 1.* (c) *Al tempo 2, i valori mostrati appaiono sugli output dei confrontatori C e D a profondità 2. I fili di output b_1 e b_4 portano il loro valore finale, diversamente dai fili di output b_2 e b_3.* (d) *Al tempo 3, i valori mostrati appaiono sugli output del confrontatore E a profondità 3. I fili di output b_2 e b_3 portano ora il loro valore finale.*

tutti i loro valori di input disponibili. Supponendo che ogni confrontatore richieda un'unità di tempo per calcolare i valori di output, i confrontatori A e B producono il loro risultato al tempo 1; i valori risultanti sono mostrati nella figura 28.2(b). Si noti che i confrontatori A e B producono i loro valori nello stesso tempo, o "in parallelo". Quindi, al tempo 1 i confrontatori C e D, ma non E, hanno tutti i loro valori di input disponibili. Un'unità di tempo dopo, al tempo 2, essi producono il loro output, come mostrato nella figura 28.2(c). Anche i confrontatori C e D operano in parallelo. L'output in alto del confrontatore C e l'output in basso del confrontatore D si connettono, rispettivamente, ai fili di output b_1 e b_4 della rete di confrontatori, questi ultimi conducono quindi i loro valori finali al tempo 2. Nel frattempo, al tempo 2, il confrontatore E ha disponibili i suoi valori di input, e la figura 28.2(d) mostra che esso produce i suoi valori di output al tempo 3. Questi valori sono condotti sulla rete ai fili di output b_2 e b_3 e la sequenza di output $\langle 2, 5, 6, 9 \rangle$ è ora completa.

Sotto l'ipotesi che ogni confrontatore richieda un'unità di tempo, si può definire il "tempo di esecuzione" di una rete di confrontatori, cioè, il tempo richiesto perché tutti i fili di output ricevano i loro valori dopo che i fili di input abbiano ricevuto i loro. Informalmente, questo tempo è il più grande numero di confrontatori che qualsiasi elemento di input possa attraversare viaggiando da un filo di input a un filo di output. Più formalmente, si definisce la ***profondità*** di un filo come segue. Un filo di input di una rete di confrontatori ha profondità 0. Ora, se un confrontatore ha due fili di input di profondità d_x e d_y, allora i suoi fili di output hanno profondità $\max(d_x, d_y) + 1$. Poiché in una rete di confrontatori non vi sono cicli di confrontatori, la profondità di un filo è ben definita e si definisce la profondità di un

confrontatore come la profondità dei suoi fili di output. La figura 28.2 mostra le profondità dei confrontatori. La profondità di una rete di confrontatori è la profondità massima di un filo di output o, equivalentemente, la profondità massima di un confrontatore. La rete di confrontatori della figura 28.2, per esempio, ha profondità 3 perché il confrontatore E ha profondità 3. Se ogni confrontatore richiede un'unità di tempo per produrre il suo valore di output e se gli input della rete appaiono al tempo 0, un confrontatore alla profondità d produce il suo output al tempo d: quindi la profondità della rete è uguale al tempo che la rete impiega per produrre i valori in tutti i suoi fili di output.

Una ***rete di ordinamento*** è una rete di confrontatori in cui la sequenza di output è monotona crescente (cioè, $b_1 \le b_2 \le \ldots \le b_n$) per *ogni* sequenza di input. Naturalmente non tutte le reti di confrontatori sono reti di ordinamento, ma la rete della figura 28.2 lo è. Per capire il perché, si osservi che dopo il tempo 1, il minimo dei quattro valori di input è stato prodotto o dall'output in alto del confrontatore A o dall'output in alto del confrontatore B. Dopo il tempo 2, perciò il minimo deve essere sull'output in alto del confrontatore C. In modo simmetrico si mostra che dopo il tempo 2 il massimo dei quattro valori di input è stato prodotto dal confrontatore D. Al confrontatore E non rimane quindi che assicurare che i due elementi medi occupino la loro giusta posizione, il che si verifica al tempo 3.

Una rete di confrontatori è come una procedura nel senso che si specifica come devono essere eseguiti i confronti, ma è diversa da una procedura nel senso che il numero di confrontatori in essa contenuti dipende dal numero di fili di input e di output. Quindi, in realtà, si descriveranno "famiglie" di reti di ordinamento. Per esempio, obiettivo di questo capitolo è di sviluppare una famiglia SORTER di reti di ordinamento efficienti. Una data rete di una famiglia è individuata con il nome della famiglia e il numero di fili di input (che è uguale al numero di fili di output). Per esempio, la rete di ordinamento con n input e n output nella famiglia SORTER è chiamata SORTER[n].

Esercizi

28.1-1 Mostrare i valori che appaiono su tutti i fili della rete della figura 28.2 quando sia data la sequenza di input $\langle 9, 6, 5, 2 \rangle$.

28.1-2 Sia n una potenza esatta di 2. Mostrare come costruire una rete di confrontatori con n input e n output di profondità $\lg n$ in cui il filo di output in alto porti sempre il minimo valore in input e quello in basso porti sempre il massimo valore in input.

28.1-3 Il professor Nielsen afferma che se si aggiunge un confrontatore ad una rete di ordinamento in una qualunque posizione, allora anche la rete che ne risulta è in grado di ordinare. Mostrare che il professore si sbaglia aggiungendo un confrontatore alla rete della figura 28.2 in modo tale che la rete risultante non ordini ogni permutazione dell'input.

28.1-4 Si dimostri che qualsiasi rete di ordinamento di n input ha profondità almeno $\lg n$.

28.1-5 Si dimostri che il numero di confrontatori in qualsiasi rete di ordinamento è almeno $\Omega(n \lg n)$.

Figura 28.3 *Una rete di ordinamento basata sull'insertion sort da usare nell'Esercizio 28.1-6.*

28.1-6 Si consideri la rete di confrontatori mostrata in figura 28.3. Si mostri che in effetti essa è una rete di ordinamento e si descriva come la sua struttura sia in relazione con quella dell'insertion sort (paragrafo 1.1).

28.1-7 Si può rappresentare una rete di confrontatori con c confrontatori come una lista di c coppie di interi nell'intervallo da 1 a n. Se due coppie contengono un intero in comune, l'ordine nella rete dei confrontatori corrispondenti è determinato dall'ordine delle coppie nella lista. Data questa rappresentazione, si descriva un algoritmo (seriale) che determini in tempo $O(n + c)$ la profondità di una rete di confrontatori.

* *28.1-8* Si supponga di introdurre, oltre al confrontatore di tipo standard, un confrontatore "decrescente" che produce il suo output minimo sul filo in basso e il suo output massimo sul filo in alto. Mostrare come trasformare qualunque rete di ordinamento che usa un numero totale c di confrontatori standard (crescenti) e decrescenti, in una che usa c confrontatori standard. Dimostrare che il metodo di trasformazione è corretto.

28.2 Il principio zero-uno

Il ***principio zero-uno*** dice che se una rete di confrontatori funziona correttamente quando ogni input è preso dall'insieme $\{0, 1\}$, allora funziona correttamente su numeri arbitrari di input. (I numeri possono essere interi, reali o, in generale, qualunque insieme di valori presi da qualunque insieme totalmente ordinato.) Dunque, nella costruzione di reti di ordinamento ed altre reti di confrontatori, il principio zero-uno permette di focalizzare l'attenzione sul loro comportamento su sequenze di input consistenti esclusivamente di 0 e 1. Dopo aver costruito una rete di ordinamento e aver provato che può ordinare tutte le sequenze zero-uno, si utilizzerà il principio zero-uno per mostrare che la rete ordina appropriatamente sequenze di valori arbitrari.

La prova del principio zero-uno si basa sulla nozione di funzione monotona crescente. (Si veda il paragrafo 2.2).

Figura 28.4 *Il comportamento del confrontatore nella dimostrazione del lemma 28.1. La funzione f è monotona crescente.*

Lemma 28.1

Se una rete di confrontatori trasforma la sequenza di input $a = \langle a_1, a_2, \ldots, a_n \rangle$ nella sequenza di output $b = \langle b_1, b_2, \ldots, b_n \rangle$, allora per qualsiasi funzione f monotona crescente, la rete trasforma la sequenza di input $f(a) = \langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$ nella sequenza di output $f(b) = \langle f(b_1), f(b_2), \ldots, f(b_n) \rangle$.

Dimostrazione. Si comincia col dimostrare l'affermazione che se f è una funzione monotona crescente, allora un confrontatore singolo con input $f(x)$ e $f(y)$ produce gli output $f(\min(x, y))$ e $f(\max(x, y))$. Quindi si userà l'induzione per dimostrare il lemma.

Per dimostrare l'affermazione, si consideri un confrontatore i cui valori di input siano x e y. L'output superiore del confrontatore è $\min(x, y)$ e l'output inferiore è $\max(x, y)$. Si supponga di applicare $f(x)$ e $f(y)$ agli input del confrontatore, come mostrato nella figura 28.4. Il comportamento del confrontatore produce il valore $\min(f(x), f(y))$ sull'output superiore e il valore $\max(f(x), f(y))$ sull'output inferiore. Poiché f è monotona crescente, $x \le y$ implica $f(x) \le f(y)$. Di conseguenza si hanno le identità

$$\min(f(x), f(y)) = f(\min(x, y)) ,$$
$$\max(f(x), f(y)) = f(\max(x, y)) .$$

Pertanto, il confrontatore produce i valori $f(\min(x, y))$ e $f(\max(x, y))$ quando i suoi input sono $f(x)$ e $f(y)$, il che completa la dimostrazione dell'affermazione.

Si può usare l'induzione sulla profondità di ogni filo in una rete di confrontatori generale per provare un risultato più forte dell'affermazione del lemma: se un filo assume il valore a_i quando la sequenza di input a è applicata alla rete, allora assumerà il valore $f(a_i)$ quando è applicata la sequenza di input $f(a)$. Poiché tale proprietà riguarda anche i fili di output, la sua dimostrazione proverà il lemma.

Per quanto riguarda la base, si consideri un filo a profondità 0, cioè un filo di input a_i. Il risultato segue banalmente: quando $f(a)$ è applicato alla rete, il filo di input porterà $f(a_i)$. Per il passo induttivo, si consideri un filo a profondità d, dove $d \ge 1$. Il filo è l'output di un confrontatore a profondità d e i fili di input di questo confrontatore sono a una profondità strettamente minore di d. Quindi dall'ipotesi induttiva, se i fili di input al confrontatore portano i valori a_i e a_j quando è applicata la sequenza di input a, allora portano $f(a_i)$ e $f(a_j)$ quando è applicata la sequenza di input $f(a)$. Dalla affermazione provata precedentemente, i fili di output di questo confrontatore portano $f(\min(a_i, a_j))$ e $f(\max(a_i, a_j))$. Poiché essi portano $\min(a_i, a_j)$ e $\max(a_i, a_j)$ quando è applicata la sequenza di input a, il lemma è dimostrato. ■

La figura 28.5 mostra, come esempio di applicazione del lemma 28.1, la rete di ordinamento della figura 28.2 con la funzione monotona crescente $f(x) = \lceil x/2 \rceil$ applicata agli input. Il valore di ogni filo è f applicata al valore dello stesso filo nella figura 28.2.

Figura 28.5 (a) *La rete di ordinamento della figura 28.2 con sequenza di input* $\langle 9, 5, 2, 6 \rangle$. (b) *La stessa rete di ordinamento con la funzione monotona crescente* $f(x) = \lceil x/2 \rceil$ *applicata agli input. Ogni filo in questa rete ha il valore di f applicata al valore del corrispondente filo in* (a).

Quando una rete di confrontatori è una rete di ordinamento, il lemma 28.1 consente di dimostrare il seguente importante risultato.

Teorema 28.2 (Principio zero-uno)

Se una rete di confrontatori con n input ordina correttamente tutte le 2^n possibili sequenze di 0 e 1, allora ordina correttamente tutte le sequenze di numeri arbitrari.

Dimostrazione. Si supponga per assurdo che la rete ordini tutte le sequenze zero-uno, ma che esista una sequenza di numeri arbitrari che la rete non ordina correttamente. Cioè esiste una sequenza di input $\langle a_1, a_2, \ldots, a_n \rangle$ contenente gli elementi a_i e a_j tali che $a_i < a_j$, ma la rete pone a_j prima di a_i nella sequenza di output. Si definisce una funzione f monotona crescente come

$$f(x) = \begin{cases} 0 & \text{se } x \le a_i , \\ 1 & \text{se } x > a_i . \end{cases}$$

Poiché la rete pone a_j prima di a_i nella sequenza di output quando l'input è $\langle a_1, a_2, \ldots, a_n \rangle$, dal lemma 28.1 segue che la stessa rete mette $f(a_j)$ prima di $f(a_i)$ nella sequenza di output quando l'input è $\langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$. Ma poiché $f(a_j) = 1$ e $f(a_i) = 0$, si ottiene la contraddizione che la rete non ordina correttamente la sequenza zero-uno $\langle f(a_1), f(a_2), \ldots, f(a_n) \rangle$. ■

Esercizi

28.2-1 Si dimostri che applicando una funzione monotona crescente a una sequenza ordinata si produce una sequenza ordinata.

28.2-2 Si dimostri che una rete di confrontatori con n input ordina correttamente la sequenza di input $\langle n, n-1, \ldots, 1 \rangle$ se e solo se ordina correttamente le sequenze zero-uno di input $\langle 1, 0, 0, \ldots, 0, 0 \rangle$, $\langle 1, 1, 0, \ldots, 0, 0 \rangle$, $\ldots$, $\langle 1, 1, 1, \ldots, 1, 0 \rangle$.

28.2-3 Si usi il principio zero-uno per dimostrare che la rete di confrontatori mostrata in figura 28.6 è una rete di ordinamento.

Figura 28.6 *Una rete di ordinamento per ordinare 4 numeri.*

28.2-4 Si definisca e si provi un analogo del principio zero-uno per un modello ad albero di decisione. (*Suggerimento:* assicurarsi di gestire in modo appropriato l'uguaglianza.)

28.2-5 Si dimostri che una rete di ordinamento con n input deve contenere almeno un confrontatore tra l'i-esima e l'$(i + 1)$-esima linea per ogni: $i = 1, 2, \ldots, n - 1$.

28.3 Una rete di ordinamento bitonico

Il primo passo nella costruzione di una rete di ordinamento efficiente è di costruire una rete di confrontatori che possa ordinare qualunque ***sequenza bitonica***: una sequenza che cresce monotonamente e poi decresce monotonamente, oppure può diventare tale mediante uno spostamento circolare. Per esempio le sequenze $\langle 1, 4, 6, 8, 3, 2\rangle$ e $\langle 9, 8, 3, 2, 4, 6\rangle$ sono entrambe bitoniche. Le sequenze zero-uno bitoniche hanno una struttura semplice. Hanno la forma $0^i1^j0^k$ oppure la forma $1^i0^j1^k$, per qualche $i, j, k \geq 0$. Si noti che anche una sequenza che sia monotona crescente o monotona decrescente è bitonica.

L'ordinatore bitonico che sarà costruito è una rete di confrontatori che ordina sequenze bitoniche di 0 e 1. L'Esercizio 28.3-6 richiede di mostrare che l'ordinatore bitonico può ordinare sequenze bitoniche di numeri arbitrari.

Half-cleaner

Un ordinatore bitonico è formato da alcuni stadi, ognuno dei quali è chiamato ***half-cleaner***. Ogni half-cleaner è una rete di confrontatori di profondità 1 in cui la linea i di input è confrontata con la linea $i + n/2$ per $i = 1, 2, \ldots, n/2$. (Si suppone che n sia pari.) La figura 28.7 mostra HALF-CLEANER[8], l'half-cleaner con 8 input e 8 output.

Quando una sequenza bitonica di 0 e 1 è applicata come input a un half-cleaner, questo produce una sequenza di output in cui i valori più piccoli sono nella metà in alto, i valori più grandi sono nella metà in basso e entrambe le metà sono bitoniche. In effetti almeno una metà è *pura* – consiste di tutti 0 o di tutti 1 – ed è da questa proprietà che deriva il nome half-cleaner (semi-purificatore). (Si noti che tutte le sequenze pure sono bitoniche.) Il prossimo lemma dimostra queste proprietà dell'half-cleaner.

Lemma 28.3

Se l'input di un half-cleaner è una sequenza bitonica di 0 e 1, allora l'output soddisfa le seguenti proprietà: sia la metà in alto che la metà in basso sono bitoniche, ogni elemento nella metà in alto è minore o uguale di ogni elemento della metà in basso e almeno una metà è pura.

Figura 28.7 *La rete di confrontatori* HALF-CLEANER[8]. *Sono mostrati due differenti esempi di input e output zero-uno. Si suppone che l'input sia bitonico. Un half-cleaner assicura che ogni elemento di output nella metà in alto sia minore o uguale di ogni elemento di output nella metà in basso. Inoltre, entrambe le metà sono bitoniche e almeno una è pura.*

Dimostrazione. La rete di confrontatori HALF-CLEANER[n] confronta gli input i e $i + n/2$ per $i = 1, 2, \ldots, n/2$. Senza perdita di generalità, si supponga che l'input sia nella forma $00\ldots011\ldots100\ldots0$. (La situazione in cui l'input è nella forma $11\ldots100\ldots011\ldots1$ è simmetrica.)

Vi sono tre possibili casi a seconda di quale sia il blocco di 0 o di 1 consecutivi in cui va a cadere il punto di mezzo $n/2$ e uno di questi casi (quello in cui il punto di mezzo cade nel blocco di 1) va suddiviso ulteriormente in due casi. I quattro casi sono mostrati nella figura 28.8. In ogni caso mostrato, il lemma è valido. ■

Ordinatore bitonico

Combinando ricorsivamente alcuni half-cleaner, come mostrato nella figura 28.9, si può costruire un ***ordinatore bitonico*** che è una rete di ordinamento per sequenze bitoniche. Il primo stadio di BITONIC-SORTER[n] è HALF-CLEANER[n] che, dal lemma 28.3, produce due sequenze bitoniche, lunghe $n/2$, tali che ogni elemento nella metà in alto è minore o uguale di ogni elemento nella metà in basso. Pertanto, si può completare l'ordinamento usando due copie di BITONIC-SORTER[$n/2$] per ordinare le due metà ricorsivamente. Nella figura 28.9(a), la ricorsione viene mostrata esplicitamente e, nella figura 28.9(b), la ricorsione è stata sviluppata per mostrare gli half-cleaner progressivamente più piccoli che costituiscono l'altra parte dell'ordinatore bitonico. La profondità $D(n)$ di BITONIC-SORTER[n] è data dalla ricorrenza

$$D(n) = \begin{cases} 0 & \text{se } n = 1 , \\ D(n/2) + 1 & \text{se } n = 2^k \text{ e } k \geq 1 , \end{cases}$$

la cui soluzione è $D(n) = \lg n$.

Pertanto una sequenza zero-uno bitonica può essere ordinata con Bitonic-Sorter che ha profondità $\lg n$. Dall'analogo al principio zero-uno, dato da dimostrare nell'Esercizio 28.3-6, segue che qualunque sequenza bitonica di numeri arbitrari può essere ordinata da questa rete.

Figura 28.8 *I possibili confronti in* HALF-CLEANER[n]. *Si assume che la sequenza di input sia una sequenza bitonica di* 0 *e* 1 *e, senza perdita di generalità, si assume che l'input sia della forma* 00…011…100…0. *Le sottosequenze di* 0 *sono bianche e le sottosequenze di* 1 *sono grigie. Si può pensare che gli* n *input siano divisi in due metà in modo che, per* $i = 1, 2, \ldots, n/2$, *siano confrontati gli input* i *e* $i + n/2$. (**a**)-(**b**) *Casi in cui la divisione cade nella sottosequenza di* 1. (**c**)-(**d**) *Casi in cui la divisione cade in una sottosequenza di zeri. In tutti i casi, ogni elemento nella metà in alto è minore o uguale di ogni elemento nella metà in basso, entrambe le metà sono bitoniche e almeno una metà è pura.*

Figura 28.9 *La rete di confrontatori* BITONIC-SORTER[n], *con* $n = 8$. (**a**) *La costruzione ricorsiva:* HALF-CLEANER[n] *seguito da due copie di* BITONIC-SORTER[$n/2$] *che operano in parallelo.* (**b**) *La rete dopo lo sviluppo della ricorsione. Ogni half-cleaner ha un sottofondo grigio. I valori da ordinare sono mostrati sui fili.*

Esercizi

28.3-1 Quante sono le sequenze bitoniche di 0 e 1 di lunghezza n?

28.3-2 Si dimostri che BITONIC-SORTER[n], dove n è una potenza esatta di 2, contiene $\Theta(n \lg n)$ confrontatori.

28.3-3 Si descriva come possa essere costruito un ordinatore bitonico di profondità $O(\lg n)$ quando il numero n di input non è una potenza esatta di 2.

28.3-4 Si dimostri che se l'input di un half-cleaner è una sequenza bitonica di numeri arbitrari, l'output soddisfa le seguenti proprietà: sia la metà in alto che la metà in basso sono bitoniche e ogni elemento nella metà in alto è piccolo almeno quanto ogni elemento della metà in basso.

28.3-5 Si considerino due sequenze di 0 e 1. Dimostrare che se ciascun elemento in una sequenza è piccolo almeno quanto ogni elemento nell'altra sequenza, allora una delle due sequenze è pura.

28.3-6 Si provi il seguente principio, analogo al principio zero-uno, per reti di ordinamento bitonico: una rete di confrontatori che può ordinare qualunque sequenza bitonica di 0 e 1 può ordinare qualunque sequenza bitonica di numeri arbitrari.

28.4 Una rete di fusione

La rete di ordinamento viene costruita a partire da ***reti di fusione*** che sono reti capaci di fondere due sequenze ordinate di input in una sequenza ordinata di output. Si modifica BITONIC-SORTER[n] per creare la rete di fusione MERGER[n]. Come con l'ordinatore bitonico, si proverà la correttezza della rete di fusione solo per input che siano sequenze zero-uno. L'Esercizio 28.4-1 richiede di mostrare come la dimostrazione possa essere estesa a valori arbitrari di input.

La rete di fusione è basata sulla seguente intuizione. Date due sequenze ordinate di input, se si inverte l'ordine della seconda sequenza e quindi si concatenano le due sequenze, la sequenza risultante è bitonica. Per esempio, date le sequenze zero-uno ordinate $X = 00000111$ e $Y = 00001111$, si inverte Y per avere $Y^R = 11110000$. Concatenando X e Y^R si ottiene 0000011111110000 che è bitonica. Pertanto, per fondere le due sequenze di input X e Y, è sufficiente eseguire un ordinamento bitonico su X concatenato con Y^R.

Si può costruire MERGER[n] modificando il primo half-cleaner di BITONIC-SORTER[n]. Il punto chiave è di eseguire l'inversione della seconda metà dell'input implicitamente. Date due sequenze ordinate $\langle a_1, a_2, \ldots, a_{n/2}\rangle$ e $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n\rangle$ da fondere, si vuole ottenere lo stesso effetto dell'ordinamento bitonico della sequenza $\langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1}\rangle$. Poiché l'half-cleaner di BITONIC-SORTER[n] confronta gli input i e $n/2 + i$, per $i = 1, 2, \ldots, n/2$, al primo stadio della rete di fusione saranno confrontati gli input i e $n - i + 1$. La figura 28.10 mostra la corrispondenza. L'unica sottigliezza è che l'ordine degli output della parte bassa del primo stadio di MERGER[n] è invertito rispetto all'ordine degli output di un comune half-cleaner. Tuttavia, poiché l'inversione di una sequenza bitonica è bitonica, gli output in alto e in basso del primo stadio della rete di fusione soddisfano le proprietà nel lemma 28.3 e pertanto le parti alta e bassa possono essere ordinate in modo bitonico in parallelo per produrre l'output ordinato delle reti di fusione.

La rete di fusione risultante è mostrata nella figura 28.11. Soltanto il primo stadio di MERGER[n] è diverso da BITONIC-SORTER[n]. Di conseguenza la profondità di MERGER[n] è $\lg n$, la stessa di BITONIC-SORTER[n].

Esercizi

28.4-1 Si provi il principio, analogo al principio zero-uno, per reti di fusione. Più precisamente, si mostri che una rete di confrontatori che può fondere due qualsiasi sequenze monotone crescenti di 0 e 1, può fondere due qualsiasi sequenze monotone crescenti di numeri arbitrari.

28.4-2 Quante sequenze zero-uno distinte devono essere applicate all'input di una rete di confrontatori per verificare che sia una rete di fusione?

28.4-3 Si mostri che qualunque rete che possa fondere un elemento con $n - 1$ elementi ordinati per produrre una sequenza ordinata di lunghezza n deve avere profondità almeno $\lg n$.

* *28.4-4* Si consideri una rete di fusione con input $a_1, a_2, \ldots, a_n$, con n potenza esatta di 2, in cui le due sequenze monotone da fondere siano $\langle a_1, a_3, \ldots, a_{n-1}\rangle$ e $\langle a_2, a_4, \ldots, a_n\rangle$. Si dimostri che il numero di confrontatori in questo tipo di rete di fusione è $\Omega(n \lg n)$. Perché questo è un interessante limite inferiore? (*Suggerimento:* si partizioni il confrontatore in tre insiemi.)

* *28.4-5* Si provi che qualsiasi rete di fusione, a prescindere dall'ordine dell'input, richiede $\Omega(n \lg n)$ confrontatori.

Figura 28.10 *Confronto tra il primo stadio di* MERGER[n] *e* HALF-CLEANER[n], *per* $n = 8$. (a) *Il primo stadio di* MERGER[n] *trasforma le due sequenze monotone di input* $\langle a_1, a_2, \ldots, a_{n/2}\rangle$ *e* $\langle a_{n/2+1}, a_{n/2+2}, \ldots, a_n\rangle$ *nelle due sequenze bitoniche* $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ *e* $\langle b_{n/2+1}, b_{n/2+2}, \ldots, b_n\rangle$. (b) *L'operazione equivalente di* HALF-CLEANER[n]. *La sequenza bitonica di input* $\langle a_1, a_2, \ldots, a_{n/2}, a_n, a_{n-1}, \ldots, a_{n/2+1}\rangle$ *è trasformata nelle due sequenze bitoniche* $\langle b_1, b_2, \ldots, b_{n/2}\rangle$ *e* $\langle b_n, b_{n-1}, \ldots, b_{n/2+1}\rangle$.

Figura 28.11 *Una rete che fonde due sequenze ordinate di input in una sequenza ordinata di output. La rete* MERGER[n] *può essere vista come* BITONIC-SORTER[n] *con il primo half-cleaner alterato per confrontare gli input* i *e* $n - i + 1$ *per* $i = 1, 2, \ldots, n/2$. *In questo caso* $n = 8$. (a) *La rete decomposta nel primo stadio seguita da due copie parallele di* BITONIC-SORTER[$n/2$]. (b) *La stessa rete con la ricorsione sviluppata. I valori zero-uno di esempio sono mostrati sui fili e gli stadi sono evidenziati in grigio.*

28.5 Una rete di ordinamento

Si hanno ora tutti gli strumenti necessari per costruire una rete che possa ordinare qualunque sequenza di input. La rete di ordinamento SORTER[n] usa la rete di fusione per realizzare una versione parallela di merge sort del paragrafo 1.3.1. La costruzione e il comportamento della rete di ordinamento sono mostrati nella figura 28.12.

La figura 28.12(a) mostra la costruzione ricorsiva di SORTER[n]. Gli n elementi di input sono ordinati usando due copie di SORTER[$n/2$] per ordinare ricorsivamente (in parallelo) due sottosequenze ognuna di dimensione $n/2$. Le due sequenze risultanti sono quindi fuse da MERGER[n]. Il caso limite per la ricorsione si ha per $n = 1$, in cui si può usare un filo per ordinare la sequenza di 1 elemento: infatti la sequenza di 1 elemento è già ordinata. La figura 28.12(b) mostra il risultato dello sviluppo della ricorsione e la figura 28.12(c) mostra la rete effettiva ottenuta sostituendo i moduli MERGER nella figura 28.12(b) con le reti di fusione effettive.

Figura 28.12 *La rete di ordinamento* SORTER[n] *costruita combinando ricorsivamente reti di fusione.* (**a**) *La costruzione ricorsiva.* (**b**) *Sviluppo della ricorsione.* (**c**) *Sostituzione dei moduli* MERGER *con le reti di fusione effettive. È indicata la profondità di ogni confrontatore e i valori zero-uno di esempio sono mostrati sui fili.*

I dati attraversano lg$n$ stadi nella rete SORTER[n]. Ogni singolo input della rete è già una sequenza ordinata di un elemento. Il primo stadio di SORTER[n] è costituito da $n/2$ copie di MERGER[2] che lavorano in parallelo per fondere coppie di sequenze di un elemento producendo sequenze ordinate di lunghezza 2. Il secondo è costituito da $n/4$ copie di MERGER[4] che fondono coppie di sequenze ordinate di 2 elementi producendo sequenze ordinate di lunghezza 4. In generale, per $k = 1, 2, \ldots, \lg n$, lo stadio k è costituito da $n/2^k$ copie di MERGER[2^k] che fondono coppie di sequenze ordinate di 2^{k-1} elementi producendo sequenze ordinate di lunghezza 2^k. Nello stadio finale è prodotta una sequenza ordinata consistente di tutti i valori di input. Si può dimostrare per induzione che questa rete di ordinamento ordina sequenze zero-uno, quindi, per il principio zero-uno (Teorema 28.2), può ordinare valori arbitrari.

Si può analizzare la profondità della rete di ordinamento in modo ricorsivo. La profondità $D(n)$ di SORTER[n] è data dalla somma tra la profondità $D(n/2)$ di SORTER[$n/2$] (le copie di SORTER[$n/2$] sono due, ma operano in parallelo) e la profondità lg$n$ di MERGER[n]. Di conseguenza la profondità di SORTER[n] è data dalla ricorrenza

$$D(n) = \begin{cases} 0 & \text{se } n = 1 , \\ D(n/2) + \lg n & \text{se } n = 2^k \text{ e } k \geq 1 , \end{cases}$$

la cui soluzione è $D(n) = \Theta(\lg^2 n)$. Pertanto si possono ordinare n numeri in parallelo in tempo $O(\lg^2 n)$.

Esercizi

28.5-1 Quanti sono i confrontatori in SORTER[n]?

28.5-2 Si dimostri che la profondità di SORTER[n] è esattamente $(\lg n)(\lg n + 1)/2$.

28.5-3 Si supponga di modificare un confrontatore in modo che prenda come input due liste ordinate di lunghezza k, le fonda e restituisca sul suo output "max" i k elementi più grandi e sul suo output "min" i k più piccoli. Si mostri che qualunque rete di ordinamento su n input con confrontatori così modificati può ordinare nk numeri, supponendo che ogni input della rete sia una lista ordinata di lunghezza k.

28.5-4 Si supponga di avere $2n$ elementi $\langle a_1, a_2, \ldots, a_{2n} \rangle$ e di volere ottenere gli n più piccoli e gli n più grandi. Si dimostri che si può fare ciò con una ulteriore profondità costante dopo aver ordinato separatamente $\langle a_1, a_2, \ldots, a_n \rangle$ e $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n} \rangle$.

* *28.5-5* Sia $S(k)$ la profondità di una rete di ordinamento con k input e sia $M(k)$ la profondità di una rete di fusione con $2k$ input. Si supponga di avere una sequenza di n numeri da ordinare e di sapere che ogni numero si trova all'interno delle k posizioni che contengono la sua posizione corretta secondo l'ordinamento. Si mostri che si possono ordinare gli n numeri con profondità $S(k) + 2M(k)$.

* *28.5-6* Si possono ordinare gli elementi di una matrice $m \times m$ ripetendo la seguente procedura k volte:

1. Si ordini ogni riga dispari in ordine monotono crescente.
2. Si ordini ogni riga pari in ordine monotono decrescente.
3. Si ordini ogni colonna in ordine monotono crescente.

Quante sono le k iterazioni di questa procedura richieste per l'ordinamento e qual è la configurazione dell'output ordinato?

Problemi

28-1 ***Reti di ordinamento per trasposizione***

Una rete di confrontatori è una ***rete di trasposizione*** se ogni confrontatore connette linee adiacenti, come nella rete della figura 28.3.

a. Si mostri che qualunque rete di trasposizione che ordini n input ha $\Omega(n^2)$ confrontatori.

b. Si dimostri che una rete di trasposizione con n input è una rete di ordinamento se e solo se ordina la sequenza $\langle n, n-1, \ldots, 1 \rangle$. (*Suggerimento:* si usi una dimostrazione per induzione analoga a quella della dimostrazione del lemma 28.1.)

Una ***rete di ordinamento pari-dispari*** su n input $\langle a_1, a_2, \ldots, a_n \rangle$ ha n livelli di confrontatori.

Figura 28.13 *Una rete di ordinamento pari-dispari su 8 input.*

La figura 28.13 mostra una rete di trasposizione pari-dispari su 8 input. Come si può vedere dalla figura per $i = 1, 2, \ldots, n$ e $d = 1, 2, \ldots, n$, la linea i è connessa tramite un confrontatore di profondità d alla linea $j = i + (-1)^{i+d}$ se $1 \le j \le n$.

c. Si dimostri che la famiglia di reti di ordinamento pari-dispari è in effetti una famiglia di reti di ordinamento.

28-2 *Reti di fusione pari-dispari di Batcher*

Nel paragrafo 28.4 si è visto come costruire una rete di fusione basata sull'ordinamento bitonico. In questo problema, si costruirà una ***rete di fusione pari-dispari***. Si supponga che n sia una potenza esatta di 2 e che si desideri fondere la sequenza ordinata di elementi sulle linee $\langle a_1, a_2, \ldots, a_n \rangle$ con gli elementi sulle linee $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n} \rangle$. Si costruiscono ricorsivamente due reti di fusione pari-dispari che fondono sottosequenze ordinate in parallelo. La prima fonde la sequenza sulle linee $\langle a_1, a_3, \ldots, a_{n-1} \rangle$ con la sequenza sulle linee $\langle a_{n+1}, a_{n+3}, \ldots, a_{2n-1} \rangle$ (gli elementi dispari). La seconda fonde $\langle a_2, a_4, \ldots, a_n \rangle$ con $\langle a_{n+2}, a_{n+4}, \ldots, a_{2n} \rangle$ (gli elementi pari). Per combinare le due sottosequenze ordinate si pone un confrontatore tra a_{2i-1} e a_{2i} per $i = 1, 2, \ldots, n$.

a. Disegnare una rete di fusione con $2n$ input per $n = 4$.

b. Si usi il principio zero-uno per dimostrare che qualsiasi rete di fusione pari-dispari di $2n$ input è in effetti una rete di ordinamento.

c. Qual è la profondità di una rete di fusione pari-dispari di $2n$ input? Qual è la sua dimensione?

28-3 *Reti di permutazione*

Una ***rete di permutazione*** su n input e n output ha dei deviatori che gli consentono di connettere i suoi input ai suoi output rispetto a una qualunque delle $n!$ possibili permutazioni. La figura 28.14(a) mostra la rete di permutazione P_2 di 2 input e 2 output, che consiste di un unico deviatore che può essere fissato sia per trasmettere i suoi input direttamente ai suoi output che per incrociarli.

a. Spiegare perché, se si sostituisce ogni confrontatore in una rete di ordinamento con il deviatore della figura 28.14(a), la rete risultante è una rete di permutazione. Cioè, per qualunque permutazione π, vi è un modo di fissare i deviatori nella rete così che l'input i sia connesso all'output $\pi(i)$.

La figura 28.14(b) mostra la costruzione ricorsiva di una rete di permutazione P_8 di 8 input e 8 output che usa due copie di P_4 e 8 deviatori. I deviatori sono stati fissati per realizzare la

Figura 28.14 *Reti di permutazione.* (a) *La rete di permutazione P_2 che consiste di un singolo deviatore che può essere fissato in uno dei due modi mostrati.* (b) *La costruzione ricorsiva di P_8 con 8 deviatori e due P_4. I deviatori e i P_4 sono fissati per realizzare la permutazione $\pi = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$.*

permutazione $\pi = \langle \pi(1), \pi(2), \ldots, \pi(8) \rangle = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$ che richiede (ricorsivamente) che il P_4 in alto realizzi $\langle 4, 2, 3, 1 \rangle$ e il P_4 in basso realizzi $\langle 2, 3, 1, 4 \rangle$.

b. Si mostri come realizzare la permutazione $\langle 5, 3, 4, 6, 1, 8, 2, 7 \rangle$ su un P_8 disegnando la disposizione dei deviatori e le permutazioni eseguite dai due P_4.

Sia n una potenza esatta di 2. Si definisca ricorsivamente P_n in termini di due $P_{n/2}$ in modo simile a come si è definito P_8.

c. Si descriva un algoritmo che opera in tempo $O(n)$ (su una comune macchina ad accesso casuale) che specifichi la disposizione di n deviatori connessi agli input e all'output di P_n e le permutazioni che devono essere eseguite da ogni $P_{n/2}$ per realizzare una qualsiasi permutazione data di n elementi. Si dimostri che l'algoritmo è corretto.

d. Quali sono la profondità e la dimensione di P_n? Quanto impiega una comune macchina ad accesso diretto per calcolare le disposizioni di tutti i deviatori, compresi quelli dei $P_{n/2}$?

e. Si spieghi perché per $n > 2$, qualunque rete di permutazione – non solo P_n – deve realizzare qualche permutazione usando due combinazioni diverse nella disposizione dei deviatori.

Note al capitolo

In Knuth [123] si può trovare una discussione sulle reti di ordinamento e la loro storia. Apparentemente esse furono progettate nel 1954 da P. N. Armstrong, R. J. Nelson e D.J. O'Connor. Nei primi anni 60, K. E. Batcher progettò la prima rete capace di fondere due sequenze di n numeri in tempo $O(\lg n)$. Usò la fusione pari-dispari (si veda il Problema 28-2), e mostrò anche come questa tecnica potrebbe essere usata per ordinare n numeri in tempo $O(\lg^2 n)$. Poco tempo dopo, progettò un ordinatore bitonico di profondità $O(\lg n)$ simile a quello presentato nel paragrafo 28.3. Knuth attribuisce il principio zero-uno a W. C. Bouricius (1954) che lo dimostrò nel contesto degli alberi di decisione.

Per molto tempo rimase aperta la questione sull'esistenza di una rete di ordinamento di profondità $O(\lg n)$. Nel 1983, è stato mostrato che la risposta è affermativa anche se in qualche modo insoddisfacente. La rete di ordinamento AKS (dal nome dei suoi progettisti Ajtai, Komlós e Szemerédi [8]) può ordinare n numeri con profondità $O(\lg n)$ usando $O(n \lg n)$ confrontatori. Sfortunatamente le costanti nascoste nella notazione O sono piuttosto grandi (molte migliaia) e pertanto la rete non può essere considerata praticabile.

Circuiti aritmetici

29

Il modello di calcolo fornito da un comune calcolatore si basa sull'ipotesi che le operazioni aritmetiche di base – addizione, sottrazione, moltiplicazione e divisione – possano essere eseguite in tempo costante. Questa astrazione è ragionevole, poiché molte operazioni di base su una macchina ad accesso diretto hanno costi paragonabili. Tuttavia, quando si progetta il circuito che realizza queste operazioni, si scopre subito che le prestazioni dipendono dalla grandezza dei numeri su cui operare. Per esempio, si impara alle scuole elementari come si addizionano due numeri naturali, espressi come numeri decimali di n cifre, in $\Theta(n)$ passi (anche se gli insegnanti di solito non danno importanza al numero di passi richiesti).

Questo capitolo presenta i circuiti che eseguono le operazioni aritmetiche. Il miglior tempo asintotico che si possa sperare di ottenere per addizionare due numeri tramite processi seriali è $\Theta(n)$. Tuttavia, con circuiti che operano in parallelo, si può fare di meglio. In questo capitolo, si progetteranno circuiti che possano eseguire velocemente l'addizione e la moltiplicazione. (La sottrazione è essenzialmente un'addizione e la divisione è demandata al Problema 29-1.) Si assumerà che tutti gli input siano numeri naturali espressi in base 2 di n bit.

Si comincia presentando nel paragrafo 29.1 i circuiti combinatori. Si vedrà come la profondità di un circuito corrisponda al suo "tempo di esecuzione". L'addizionatore completo, che è un blocco di base per la costruzione di molti circuiti di questo capitolo, sarà usato come esempio iniziale di circuito combinatorio. Il paragrafo 29.2 presenta due circuiti combinatori per l'addizione: l'addizionatore con propagazione del riporto che impiega tempo $\Theta(n)$ e l'addizionatore con previsione del riporto che richiede solo tempo $O(\lg n)$. È presentato anche l'addizionatore senza riporto che può ridurre, in tempo $O(1)$, il problema di sommare tre numeri al problema di sommare due numeri. Il paragrafo 29.3 presenta due moltiplicatori combinatori: il moltiplicatore a griglia che impiega tempo $\Theta(n)$ e il moltiplicatore ad albero di Wallace che richiede solo tempo $\Theta(\lg n)$. Infine nel paragrafo 29.4 si presentano i circuiti con elementi di memoria temporizzati (registri) e si mostra come si possa risparmiare l'hardware riutilizzando circuiti combinatori.

29.1 Circuiti combinatori

Come le reti di confrontatori del Capitolo 28, i circuiti combinatori operano in *parallelo*: molti elementi possono calcolare valori simultaneamente in un unico passo. In questo paragrafo, si definiscono i circuiti combinatori e si studia come circuiti combinatori più grandi possano essere costruiti a partire da porte elementari.

x	$\neg x$
0	1
1	0

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

x	y	$\neg(x \wedge y)$
0	0	1
0	1	1
1	0	1
1	1	0

x	y	$\neg(x \vee y)$
0	0	1
0	1	0
1	0	0
1	1	0

Figura 29.1 *Sei porte logiche di base, con input e output binari. Sotto ogni porta vi è la tabella di verità che ne descrive il comportamento.* (a) *La porta* NOT. (b) *La porta* AND. (c) *La porta* OR. (d) *La porta* XOR (OR *esclusivo*). (e) *La porta* NAND (NOT-AND). (f) *La porta* NOR (NOT-OR).

Elementi combinatori

I circuiti aritmetici dei calcolatori reali sono costruiti a partire da elementi combinatori interconnessi tramite fili. Un ***elemento combinatorio*** è un qualunque elemento di circuito che abbia un numero costante di input e output e che esegua una funzione ben precisa. Alcuni degli elementi che saranno presentati in questo capitolo sono ***elementi combinatori booleani*** – i loro input e i loro output sono presi dall'insieme $\{0, 1\}$, dove 0 rappresenta FALSE e 1 rappresenta TRUE.

Un elemento combinatorio booleano che calcola una semplice funzione booleana è chiamato ***porta logica***. La figura 29.1 mostra le quattro porte logiche di base che saranno usate come elementi combinatori in questo capitolo: la ***porta NOT*** (o ***invertitore***), la ***porta AND***, la ***porta OR*** e la ***porta XOR***. (Sono mostrate anche altre due porte logiche - la ***porta NAND*** e la ***porta NOR*** – che sono richieste in qualche esercizio.) La porta NOT prende un unico ***input*** binario x, il cui valore è 0 o 1, e produce un ***output*** binario z il cui valore è l'opposto del valore in input. Ciascuna delle altre porte logiche richiede due input binari x e y e produce un unico output binario z.

Il comportamento di ogni porta, e di qualunque elemento booleano, può essere descritto da una ***tabella di verità***, mostrata sotto ogni porta nella figura 29.1. Una tabella di verità fornisce gli output degli elementi combinatori per ogni possibile input. Per esempio, la tabella di verità per la porta XOR dice che quando gli input sono $x = 0$ e $y = 1$, il valore di output è $z = 1$; essa calcola l'"OR esclusivo" dei due input. Si usa il simbolo $\neg$ per denotare la funzione NOT, $\wedge$ per denotare la funzione AND, $\vee$ per denotare la funzione OR e $\oplus$ per denotare la funzione XOR. Quindi, per esempio, $0 \oplus 1 = 1$.

Gli elementi combinatori dei circuiti reali non operano istantaneamente. Quando i valori di input che entrano in un elemento combinatorio diventano ***stabili*** – cioè mantengono lo stesso valore per un tempo sufficientemente lungo – si è sicuri che il valore di output degli elementi diventa stabile e corretto dopo una quantità fissata di tempo. Questa differenza di tempo si chiama ***ritardo di propagazione*** dell'elemento. Si assume in questo capitolo che tutti gli elementi combinatori abbiano ritardo di propagazione costante.

Circuiti combinatori

Un ***circuito combinatorio*** consiste di uno o più elementi combinatori interconnessi in modo aciclico. Le interconnessioni sono chiamate ***fili***. Un filo può connettere l'output di un elemento all'input di un altro, fornendo così il valore di output del primo elemento come valore di input del secondo. Sebbene un singolo filo non possa essere connesso a più di un output di elementi combinatori, esso può fornire l'input a più di un elemento combinatorio. Il numero di input di elementi connessi ad un filo è chiamato ***numero dei collegamenti di uscita del filo***. Se nessun output di un elemento è connesso a un filo, il filo è un ***input del circuito*** che accetta valori di input dall'esterno. Se nessun input di elemento è connesso a un filo, il filo è un ***output del circuito*** che fornisce i risultati del calcolo del circuito al mondo esterno. (Un filo interno può collegarsi in uscita sull'output del circuito.) I circuiti combinatori non contengono cicli e non hanno elementi di memoria (come i registri descritti nel paragrafo 29.4).

Addizionatori completi

La figura 29.2 mostra, ad esempio, un circuito combinatorio, chiamato ***addizionatore completo***, che ha in input tre bit x, y, z e restituisce due bit s e c in accordo alla seguente tabella di verità:

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

L'output s è la ***parità*** dei bit in input.

$$s = \text{parità}(x, y, z) = x \oplus y \oplus z\,, \tag{29.1}$$

Figura 29.2 *Un addizionatore completo.* (a) *Al tempo 0 i bit in input mostrati appaiono sui tre fili di input.* (b) *Al tempo 1 i valori mostrati appaiono sull'output delle porte A-D che sono a profondità 1.* (c) *Al tempo 2 i valori mostrati appaiono sull'output delle porte E ed F che sono a profondità 2.* (d) *Al tempo 3 la porta G produce il suo output che è anche l'output del circuito.*

e l'output c è la ***maggioranza*** dei bit in input,

$$c = \text{maggioranza}(x, y, z) = (x \wedge y) \vee (y \wedge z) \vee (x \wedge z) \tag{29.2}$$

(In generale, le funzioni di parità e di maggioranza possono avere un numero qualunque di bit in input. La parità è 1 se e solo se gli 1 in input sono in numero dispari. La maggioranza è 1 se e solo se più della metà dei bit in input è 1.) Si noti che i bit c ed s, presi insieme, danno la somma di x, y e z. Per esempio se $x = 1$, $y = 0$, $z = 1$, allora $\langle c, s\rangle = \langle 10\rangle$[1] che è la rappresentazione binaria di 2, la somma di x, y e z.

Ciascun input x, y e z dell'addizionatore completo ha 3 collegamenti di uscita. Quando l'operazione eseguita da un elemento combinatorio è commutativa e associativa rispetto ai suoi input (come le funzioni AND, OR, e XOR) il numero di input si dice ***numero di collegamenti di ingresso dell'elemento***. Sebbene il numero di collegamenti di ingresso di ogni porta nella figura 29.2 sia 2, si potrebbe riprogettare l'addizionatore completo sostituendo le porte XOR A ed E con un'unica porta XOR a 3 input e le porte OR F e G con un'unica porta OR a 3 input.

[1] Per chiarezza si omettono le virgole tra gli elementi della sequenza quando questi sono bit.

Per esaminare come funziona l'addizionatore completo, si assuma che ogni porta richieda un'unità di tempo. La figura 29.2(a) mostra un insieme di input che diventano stabili al tempo 0. Le porte A-D, e nessun'altra, hanno tutti i loro valori di input stabili in quel momento e quindi producono i valori mostrati nella figura 29.2(b) al tempo 1. Si noti che le porte A-D funzionano in parallelo. Le porte E ed F, ma non la G, hanno gli input stabili al tempo 1 e producono i valori mostrati nella figura 29.2(c) al tempo 2. L'output della porta E è il bit s e così l'output s dell'addizionatore è pronto al tempo 2. Però l'output c non è ancora pronto. Infine, la porta G ha input stabile al tempo 2 e produce il suo output c, mostrato in figura 29.2(d), al tempo 3.

Profondità dei circuiti

Come nel caso delle reti di confrontatori discusse nel Capitolo 28, si misura il ritardo di propagazione di un circuito combinatorio in termini del più grande numero di elementi combinatori su qualunque cammino dagli input agli output. Più precisamente, si definisce la ***profondità*** di un circuito, che corrisponde al suo "tempo di esecuzione" nel caso peggiore, induttivamente rispetto alle profondità dei suoi fili di interconnessione. La profondità di un filo in input è 0. Se un elemento combinatorio ha gli input $x_1, x_2, \ldots, x_n$ alle profondità $d_1, d_2, \ldots, d_n$ rispettivamente, allora i suoi output hanno profondità $\max\{d_1, d_2, \ldots, d_n\} + 1$. La profondità di un elemento combinatorio è la profondità dei suoi input. La profondità di un circuito combinatorio è la profondità massima di qualunque elemento combinatorio. Poiché non sono previsti circuiti combinatori contenenti cicli, le precedenti nozioni di profondità sono ben definite.

Se ogni elemento combinatorio richiede tempo costante per calcolare i suoi valori di output, allora il ritardo di propagazione attraverso un circuito combinatorio, nel caso peggiore, è proporzionale alla sua profondità. La figura 29.2 mostra la profondità di ogni porta nell'addizionatore completo. Poiché la porta con profondità più grande è la porta G, anche l'addizionatore completo ha profondità 3 che è proporzionale al tempo richiesto dal circuito per eseguire la sua funzione nel caso peggiore.

Talvolta un circuito combinatorio può essere più veloce della sua profondità. Si supponga che un sottocircuito grande fornisca un input ad uno dei due input di una porta AND ma che l'altro input della porta AND abbia valore 0. L'output della porta sarà 0, indipendentemente dall'input proveniente dal sottocircuito grande. In generale, però, non si può contare sul fatto che input specifici siano applicati al circuito, pertanto l'astrazione di profondità come "tempo di esecuzione" del circuito è abbastanza ragionevole.

Dimensione dei circuiti

Oltre alla profondità, vi è un'altra risorsa che tipicamente si desidera minimizzare quando si progetta un circuito. La ***dimensione*** di un circuito combinatorio è il numero di elementi combinatori che contiene. Intuitivamente, la dimensione del circuito corrisponde allo spazio

di memoria usato da un algoritmo. Per esempio l'addizionatore completo della figura 29.2 ha dimensione 7 poiché usa 7 porte.

Questa definizione di dimensione di un circuito non è particolarmente utile per circuiti piccoli. Dopo tutto, dato che un addizionatore completo ha un numero costante di input e output e calcola una funzione ben definita, esso soddisfa la definizione di elemento combinatorio. Un addizionatore completo costruito usando un solo elemento combinatorio ha, quindi, dimensione 1. In realtà, rispetto a questa definizione, *qualsiasi* elemento combinatorio ha dimensione 1.

La definizione di dimensione di un circuito va applicata a famiglie di circuiti che calcolano funzioni simili. Per esempio, si vedrà nel prossimo paragrafo un circuito addizionatore che prende due input di n bit. In effetti in questo caso non si parla di un unico circuito, ma piuttosto di una famiglia di circuiti – una per ogni dimensione dell'input. In questo contesto la definizione di dimensione dell'input assume maggior significato. Consente di definire appropriati elementi di circuito senza influenzare la dimensione di una qualunque realizzazione del circuito, se non per un fattore costante. Naturalmente, in pratica, le misurazioni della dimensione sono molto più complicate; esse coinvolgono non solo la scelta degli elementi combinatori, ma anche aspetti quali l'area che il circuito richiede quando viene integrato su una piastrina di silicio.

Esercizi

29.1-1 Nella figura 29.2, si sostituisca l'input y con il valore 1. Si mostri il valore risultante di ogni filo.

29.1-2 Si mostri come costruire un circuito per la parità di n input con $n-1$ porte XOR di profondità $\lceil \lg n \rceil$.

29.1-3 Si mostri che qualunque elemento combinatorio booleano può essere costruito con un numero costante di porte AND, OR e NOT. (*Suggerimento:* si realizzi la tabella di verità dell'elemento.)

29.1-4 Si mostri che qualunque funzione booleana può essere costituita esclusivamente di porte NAND.

29.1-5 Si costruisca un circuito combinatorio che esegua la funzione or esclusivo usando solo 4 porte NAND con 2 input.

29.1-6 Sia C un circuito combinatorio con n input e n output di profondità d. Se due copie di C sono connesse in serie qual è la massima profondità possibile di questo circuito a cascata? Qual è la minima profondità possibile?

29.2 Addizionatori

Si considererà ora il problema di addizionare numeri rappresentati in binario. Si presentano tre circuiti combinatori. Prima si esamina un addizionatore con propagazione del riporto che

può sommare due numeri di n bit in tempo $\Theta(n)$ usando un circuito di dimensione $\Theta(n)$. Questo tempo può essere abbassato a $O(\lg n)$ usando un addizionatore con previsione del riporto, anch'esso di dimensione $\Theta(n)$. Infine, si presenta un addizionatore senza riporto che in tempo $O(1)$ può ridurre la somma di 3 numeri di n bit alla somma di un numero di n bit e di un numero di $n+1$ bit. Il circuito ha dimensione $\Theta(n)$.

29.2.1 Addizione con propagazione del riporto

Cominciamo con il comune metodo di somma dei numeri binari. Si assume che un intero non negativo a sia rappresentato in binario da una sequenza di n bit $\langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, dove $n \geq \lceil \lg(a+1) \rceil$ e

$$a = \sum_{i=0}^{n-1} a_i 2^i .$$

Dati due numeri di n bit $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ e $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, si desidera produrne la somma $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$ di $(n+1)$ bit. La figura 29.3 mostra un esempio di addizione di due numeri di 8 bit. Si sommano le colonne da destra a sinistra, propagando il riporto dalla colonna i alla colonna $i+1$, per $i = 1, 2, \ldots, n-1$. Nella i-esima posizione, si prendono come input i bit a_i e b_i e un ***bit di riporto di ingresso*** c_i e si produce un ***bit di somma*** s_i e un bit di ***riporto di uscita*** c_{i+1}. Il bit di riporto di uscita c_{i+1} della i-esima posizione è il bit di riporto di ingresso della $(i+1)$-esima posizione. Poiché non vi è riporto di ingresso per la posizione 0, si assume che $c_0 = 0$. Il riporto di uscita c_n è il bit s_n della somma.

Si osservi che ogni bit di somma s_i è la parità dei bit a_i, b_i, e c_i (si veda l'equazione (29.1)). Inoltre, il bit di riporto di uscita c_{i+1} è il bit di maggioranza di a_i, b_i, e c_i (si veda l'equazione (29.2)). Pertanto ogni stadio della somma può essere eseguito da un addizionatore completo.

Un ***addizionatore con propagazione del riporto*** di n bit è costruito mettendo in cascata n addizionatori completi $FA_0, FA_1, \ldots, FA_{n-1}$, fornendo il riporto di uscita c_{i+1} di FA_i direttamente al riporto di ingresso di FA_{i+1}. La figura 29.4 mostra un addizionatore con propagazione del riporto di n bit. I bit di riporto si propagano da destra verso sinistra. Il riporto di ingresso c_0 dell'addizionatore completo è ***cablato*** a 0, cioè, è 0 a prescindere dai valori degli altri input. L'output è il numero $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$ di $n+1$ bit, dove s_n è uguale a c_n, il bit di riporto di uscita dell'addizionatore completo FA_n.

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 *Addizione di due numeri di 8 bit $a = \langle 01011110 \rangle$ e $b = \langle 11010101 \rangle$ per produrre la somma $s = \langle 100110011 \rangle$ di 9 bit. Ogni bit c_i è un bit di riporto. Ogni colonna di bit rappresenta, dall'alto verso il basso, c_i, a_i, b_i ed s_i per qualche i. Il riporto di ingresso c_0 è sempre 0.*

Figura 29.4 *Un addizionatore con propagazione del riporto che esegue la somma della figura 29.3. Il bit di riporto c_0, indicato con il rombo, è cablato a 0 e i bit di riporto si propagano da destra verso sinistra.*

Poiché i bit di riporto si propagano attraverso tutti gli n addizionatori completi, il tempo richiesto da un addizionatore con propagazione del riporto di n bit è $\Theta(n)$. Più precisamente, l'addizionatore completo FA_i è a profondità $i + 1$ nel circuito. Poiché FA_{n-1} è a profondità maggiore di qualunque addizionatore completo nel circuito, la profondità dell'addizionatore è n. La dimensione del circuito è $\Theta(n)$ perché esso contiene n elementi combinatori.

29.2.2 Addizione con previsione del riporto

L'addizione con propagazione del riporto richiede tempo $\Theta(n)$ a causa della propagazione dei bit del riporto attraverso il circuito. L'addizionatore con previsione del riporto evita questo ritardo di tempo $\Theta(n)$ accelerando il calcolo dei riporti mediante un circuito ad albero. Un addizionatore con previsione del riporto può sommare due numeri di n bit in tempo $O(\lg n)$.

L'osservazione chiave è che nella somma con propagazione del riporto, per $i \geq 1$, l'addizionatore completo FA_i ha due dei valori di input, a_i e b_i, pronti prima che il riporto di ingresso c_i sia pronto. L'idea dell'addizionatore con previsione del riporto è di sfruttare queste informazioni parziali.

Per esempio, sia $a_{i-1} = b_{i-1}$. Poiché il riporto di uscita c_i è la funzione di maggioranza, si ha $c_i = a_{i-1} = b_{i-1}$ *a prescindere dal riporto di ingresso* c_{i-1}. Se $a_{i-1} = b_{i-1} = 0$, si può ***eliminare*** il riporto di uscita c_i forzandolo a 0 senza aspettare che il valore c_{i-1} sia calcolato. Analogamente, se $a_{i-1} = b_{i-1} = 1$, si può ***generare*** il riporto di uscita $c_i = 1$, indipendentemente dal valore di c_{i-1}.

Tuttavia, se $a_{i-1} \neq b_{i-1}$ allora c_i dipende di c_{i-1}. In particolare $c_i = c_{i-1}$, perché il riporto di ingresso c_{i-1} è il voto decisivo nell'elezione maggioritaria che determina c_i. In questo caso si ***propaga*** il riporto poiché il riporto di uscita è il riporto di ingresso.

La figura 29.5 riassume queste relazioni in termini di ***stati del riporto*** dove k è il "riporto eliminato", g è il "riporto generato" e p è il "riporto propagato".

a_{i-1}	b_{i-1}	c_i	stati del riporto
0	0	0	k
0	1	c_{i-1}	p
1	0	c_{i-1}	p
1	1	1	g

Figura 29.5 *Il riporto di uscita c_i e lo stato del riporto corrispondenti agli input a_{i-1}, b_{i-1} e c_{i-1} dell'addizionatore completo FA_{i-1} nella somma con propagazione del riporto.*

		FA_i		
	$\otimes$	k	p	g
	k	k	k	g
FA_{i-1}	p	k	p	g
	g	k	g	g

Figura 29.6 *Lo stato del riporto della combinazione degli addizionatori completi FA_{i-1} e FA_i in termini dei loro stati individuali del riporto, dato dall'operatore di stato del riporto $\otimes$ sul dominio* {k, p, g}.

Si considerino due addizionatori completi consecutivi FA_{i-1} e FA_i insieme come un'unità combinata. Il riporto in ingresso all'unità sia c_{i-1} e il riporto di uscita sia c_{i+1}. Si può notare che l'unità combinata elimina, genera o propaga riporti come un singolo addizionatore completo. L'unità combinata elimina il suo riporto se FA_i elimina il suo riporto oppure se FA_{i-1} elimina il suo riporto e FA_i lo propaga. Analogamente, l'unità combinata genera un riporto se FA_i genera un riporto oppure se FA_{i-1} genera un riporto e FA_i lo propaga. L'unità combinata propaga il riporto, ponendo $c_{i+1} = c_{i-1}$, se entrambi gli addizionatori completi propagano i riporti. La tabella in figura 29.6 riassume come gli stati del riporto siano combinati quando si connettono i due addizionatori completi. Si può vedere questa tabella come la definizione dell'***operatore di stato del riporto*** $\otimes$ sul dominio {k, p, g}. Una importante proprietà di questo operatore è che esso è associativo, come l'Esercizio 29.2-2 richiede di verificare.

Si può usare l'operatore di stato del riporto per esprimere ogni bit di riporto c_i in termini dei suoi input. Si comincia definendo $x_0 =$ k e

$$x_i = \begin{cases} \text{k} & \text{se } a_{i-1} = b_{i-1} = 0 \ , \\ \text{p} & \text{se } a_{i-1} \neq b_{i-1} \ , \\ \text{g} & \text{se } a_{i-1} = b_{i-1} = 1 \ , \end{cases} \tag{29.3}$$

per $i = 1, 2, \ldots, n$. Pertanto, per $i = 1, 2, \ldots, n$, il valore di x_i è lo stato del riporto descritto nella figura 29.5.

Il riporto di uscita c_i di un dato addizionatore completo FA_{i-1} può dipendere dallo stato del riporto di ogni FA_j per $j = 0, 1, \ldots, i-1$. Si definisce $y_0 = x_0 =$ k e

$$\begin{aligned} y_i &= y_{i-1} \otimes x_i \\ &= x_0 \otimes x_1 \otimes \cdots \otimes x_i \end{aligned} \tag{29.4}$$

per $i = 1, 2, \ldots, n$. Si può pensare a y_i come a un "prefisso" del "prodotto" $x_0 \otimes x_1 \otimes \cdots \otimes x_n$; il processo di calcolo dei valori $y_0, y_1, \ldots, y_n$ è chiamato ***calcolo dei prefissi***. (Il Capitolo 30 discute il calcolo dei prefissi in un contesto parallelo più generale.) La figura 29.7 mostra i valori di x_i e y_i corrispondenti alla addizione binaria mostrata nella figura 29.3. Il seguente lemma fornisce il significato dei valori y_i per la somma con previsione del riporto.

Lemma 29.1

Siano $x_0, x_1, \ldots, x_n$ e $y_0, y_1, \ldots, y_n$ definite dalle equazioni (29.3) e (29.4). Per $i = 0, 1, \ldots, n$ valgono le seguenti condizioni:

1. $y_i =$ k implica $c_i = 0$.

i	8	7	6	5	4	3	2	1	0
a_i		0	1	0	1	1	1	1	0
b_i		1	1	0	1	0	1	0	1
x_i	p	g	k	g	p	g	p	p	k
y_i	g	g	k	g	g	g	k	k	k
c_i	1	1	0	1	1	1	0	0	0

Figura 29.7 *I valori di x_i e y_i per $i = 1, 2, \ldots, 8$ che corrispondono ai valori di a_i, b_i e c_i nel problema della somma binaria della figura 29.3. Ogni valore di x_i è su un sottofondo grigio insieme con i valori di a_i e b_i da cui dipende.*

2. $y_i = \mathrm{g}$ implica $c_i = 1$ e
3. $y_i = \mathrm{p}$ non si verifica.

Dimostrazione. La dimostrazione è per induzione su i. Per la base $i = 0$, si ha $y_0 = x_0 = \mathrm{k}$ per definizione e anche $c_0 = 0$. Per il passo induttivo, si assuma che il lemma valga per $i - 1$. Vi sono tre casi a seconda del valore di y_i.

1. Se $y_i = \mathrm{k}$, allora poiché $y_i = y_{i-1} \otimes x_i$, la definizione dell'operatore $\otimes$ di stato del riporto della figura 29.6 implica che $x_i = \mathrm{k}$ oppure che $x_i = \mathrm{p}$ e $y_{i-1} = \mathrm{k}$. Se $x_i = \mathrm{k}$, allora l'equazione (29.3) implica che $a_{i-1} = b_{i-1} = 0$ e pertanto $c_i =$ maggioranza $(a_{i-1}, b_{i-1}, c_{i-1}) = 0$. Se $x_i = \mathrm{p}$ e $y_{i-1} = \mathrm{k}$, allora $a_{i-1} \neq b_{i-1}$ e, per induzione, $c_{i-1} = 0$. Quindi, maggioranza $(a_{i-1}, b_{i-1}, c_{i-1}) = 0$ e così $c_i = 0$.
2. Se $y_i = \mathrm{g}$, allora si ha che $x_i = \mathrm{g}$ oppure che $x_i = \mathrm{p}$ e $y_{i-1} = \mathrm{g}$. Se $x_i = \mathrm{g}$, allora $a_{i-1} = b_{i-1} = 1$, che implica $c_i = 1$. Se $x_i = \mathrm{p}$ e $y_{i-1} = \mathrm{g}$, allora $a_{i-1} \neq b_{i-1}$ e, per induzione, $c_{i-1} = 1$ che implica $c_i = 1$.
3. Se $y_i = \mathrm{p}$, allora la figura 29.6 implica $y_{i-1} = \mathrm{p}$, che contraddice l'ipotesi induttiva. ■

Il lemma 29.1 implica la possibilità di calcolare ogni bit di riporto c_i calcolando ogni stato del riporto y_i. Dopo aver ottenuto tutti i bit di riporto, si può calcolare l'intera somma in tempo $\Theta(1)$ calcolando in parallelo i bit della somma $s_i =$ parità(a_i, b_i, c_i) per $i = 0, 1, \ldots, n$ (prendendo $a_n = b_n = 0$). Pertanto, il problema di sommare velocemente due numeri si riduce al calcolo dei prefissi $y_0, y_1, \ldots, y_n$ degli stati del riporto.

Calcolo degli stati del riporto con un circuito parallelo per il calcolo dei prefissi

Usando un circuito per il calcolo dei prefissi che opera in parallelo, invece di un circuito con propagazione del riporto che produce i suoi output uno alla volta, si possono calcolare tutti gli stati del riporto $y_0, y_1, \ldots, y_n$ più velocemente. In particolare, si progetterà un circuito parallelo per il calcolo dei prefissi di profondità $O(\lg n)$. Il circuito ha dimensione $\Theta(n)$, asintoticamente la stessa quantità di hardware di un addizionatore con propagazione del riporto.

Prima di costruire il circuito parallelo per il calcolo dei prefissi, si introduce una notazione che faciliti la comprensione di come operano i circuiti. Per gli interi i e j nell'intervallo $0 \le i \le j \le n$, si definisce

$$[i, j] = x_i \otimes x_{i+1} \otimes \cdots \otimes x_j .$$

Figura 29.8 *Organizzazione di un circuito parallelo per il calcolo dei prefissi. Il nodo mostrato è la radice di un sottoalbero le cui foglie ricevono i valori di input da x_1 a x_n. Il nodo consiste di due elementi $\otimes$ che operano in tempi diversi durante il funzionamento del circuito. Un elemento calcola $[i, k] \leftarrow [i, j-1] \otimes [j, k]$ e l'altro elemento calcola $[0, j-1] \leftarrow [0, i-1] \otimes [i, j-1]$. I valori calcolati sono mostrati sui fili.*

Pertanto, per $i = 0, 1, \ldots, n$, si ha $[i, i] = x_i$, poiché la composizione di un unico stato del riporto x_i è esso stesso. Per i, j e k che soddisfano $0 \le i < j \le k \le n$, si ha anche l'identità

$$[i, k] = [i, j-1] \otimes [j, k] , \tag{29.5}$$

poiché l'operatore di stato del riporto è associativo. Utilizzando questa notazione, l'obiettivo di un calcolo dei prefissi è, dunque, calcolare $y_i = [0, i]$ per $i = 0, 1, \ldots, n$.

L'unico elemento combinatorio usato nel circuito parallelo per il calcolo dei prefissi è un circuito che si comporta come l'operatore $\otimes$. La figura 29.8 mostra come coppie di elementi $\otimes$ siano organizzate per formare i nodi interni di un albero binario completo e la figura 29.9 illustra il circuito parallelo per il calcolo dei prefissi per $n = 8$. Si noti che i fili nel circuito seguono la struttura di un albero, ma il circuito non è un albero benché sia puramente combinatorio. Gli input $x_1, x_2, \ldots, x_n$ sono collegati sulle foglie e l'input x_0 è collegato sulla radice. Gli output $y_0, y_1, \ldots, y_{n-1}$ sono prodotti sulle foglie e l'output y_n è prodotto sulla radice. (Per facilitare la comprensione del calcolo dei prefissi, nella figure 29.8 e 29.9 gli indici delle variabili crescono da sinistra a destra piuttosto che da destra a sinistra come nelle altre figure di questo paragrafo.)

I due elementi $\otimes$ in ogni nodo tipicamente funzionano con tempi diversi e hanno profondità diverse nel circuito. Come mostrato nella figura 29.8, se il sottoalbero radicato in un dato nodo abbraccia un intervallo di input $x_i, x_{i+1}, \ldots, x_k$, il suo sottoalbero sinistro abbraccia l'intervallo

Figura 29.9 *Un circuito parallelo per il calcolo dei prefissi per n = 8.* (a) *La struttura globale del circuito e i valori portati da ogni filo.* (b) *Lo stesso circuito con valori corrispondenti alle figure* 29.3 *e* 29.7.

$x_i, x_{i+1}, \ldots, x_{j-1}$ e il suo sottoalbero destro abbraccia l'intervallo $x_j, x_{j+1}, \ldots, x_k$ allora il nodo deve generare per suo padre il prodotto $[i, k]$ di tutti gli input abbracciati dal suo sottoalbero. Poiché si può assumere induttivamente che i figli sinistro e destro del nodo generino i prodotti $[i, j-1]$ e $[j, k]$, il nodo usa semplicemente uno dei suoi elementi per calcolare $[i, k] \leftarrow [i, j-1] \otimes [j, k]$.

Qualche tempo dopo questa fase di calcolo "verso l'alto", il nodo riceve da suo padre il prodotto $[0, i-1]$ di tutti gli input che esso abbraccia e che vengono prima dell'input x_i più a sinistra. Il nodo ora calcola allo stesso modo i valori per i suoi figli. L'input più a sinistra abbracciato dal figlio sinistro del nodo è ancora x_i e così il nodo passa il valore $[0, i-1]$ al figlio sinistro immutato. L'input più a sinistra abbracciato dal figlio destro è x_j, così il nodo deve produrre $[0, j-1]$. Poiché il nodo riceve il valore $[0, i-1]$ da suo padre e il valore $[i, j-1]$ dal figlio sinistro, esso calcola semplicemente $[0, j-1] \leftarrow [0, i-1] \otimes [i, j-i]$ e manda questo valore al figlio destro.

La figura 29.9 mostra il circuito risultante, incluso il caso limite che sorge per la radice. Il valore $x_0 = [0, 0]$ è fornito come input alla radice e viene usato un altro elemento $\otimes$ per calcolare (in generale) il valore $y_n = [0, n] = [0, 0] \otimes [1, n]$.

Figura 29.10 *La costruzione di un addizionatore con previsione del riporto di n bit mostrato qui per n = 8. Esso consiste di n + 1 scatole KPG_i per i = 0, 1, ..., n. Ogni scatola KPG_i prende gli input esterni a_i e b_i (dove a_n e b_n sono cablati a 0, come indicato dal rombo) e calcola lo stato del riporto x_{i+1}. Questi valori sono caricati nel circuito parallelo che restituisce i risultati y_i del calcolo dei prefissi. Ogni unità KPG_i prende ora y_i come input, lo interpreta come il riporto di ingresso c_i e quindi restituisce il bit di somma s_i = parità(a_i, b_i, c_i). Sono mostrati valori di esempio corrispondenti a quelli mostrati nelle figure* 29.3 *e* 29.9.

Se n è una potenza esatta di 2, allora il circuito parallelo per il calcolo dei prefissi usa $2n - 1$ elementi $\otimes$. Richiede solo tempo $O(\lg n)$ per calcolare tutti gli $n + 1$ prefissi, poiché il calcolo procede verso l'alto e poi verso il basso. L'Esercizio 29.2-5 studia la profondità del circuito in maggior dettaglio.

Completamento dell'addizionatore con previsione del riporto

Ora che si ha un circuito parallelo per il calcolo dei prefissi, si può completare la descrizione dell'addizionatore con previsione del riporto. La figura 29.10 mostra la costruzione. Un ***addizionatore con previsione del riporto*** di n bit è composto di $n + 1$ ***scatole KPG***, ciascuno di dimensione $\Theta(1)$, e un circuito parallelo per il calcolo dei prefissi con input $x_0, x_1, \ldots, x_n$ (x_0 è cablato a k) e output $y_0, y_1, \ldots, y_n$. La scatola KPG_i prende gli input esterni a_i e b_i e produce il bit di somma s_i. (Gli input a_n e b_n sono cablati a 0.) Dati a_i e b_i, la scatola KPG_{i-1} calcola $x_i \in \{k, p, g\}$ secondo l'equazione (29.3) e manda questo valore come input esterno x_i del circuito parallelo per il calcolo dei prefissi. (Il valore di x_{n-1} è ignorato.) Il calcolo di tutti gli x_i richiede tempo $\Theta(1)$. Dopo un ritardo di $O(\lg n)$, il circuito parallelo per il calcolo dei prefissi produce $y_0, y_1, \ldots, y_n$. Dal lemma 29.1, y_i è k o g; non può essere p. Ogni valore y_i indica il riporto di ingresso dell'addizionatore completo FA_i nell'addizionatore con propagazione del riporto: y_i = k implica $c_i = 0$ e y_i = g implica $c_i = 1$. Quindi, il valore di y_i è caricato in KPG_i per indicare il riporto di ingresso c_i e il bit di somma s_i = parità(a_i, b_i, c_i) è prodotto in tempo costante. Pertanto, l'addizionatore con previsione del riporto richiede tempo $O(\lg n)$ e ha dimensione $\Theta(n)$.

29.2.3 Addizionatore senza riporto

Un addizionatore con previsione del riporto può sommare due numeri di n bit in tempo $O(\lg n)$. Forse può sorprendere che la somma di tre numeri di n bit richieda solo una quantità costante

8	7	6	5	4	3	2	1	0		i
	0	0	0	1	1	1	0	1	=	x
	1	1	0	0	0	1	0	1	=	y
	1	0	0	1	0	1	1	0	=	z
	0	1	0	0	1	1	1	0	=	u
1	0	0	1	0	1	0	1	0	=	v

(a)

(b)

Figura 29.11 (a) *Somma senza riporto. Dati tre numeri di n bit x, y e z, si producono un numero di n bit u e un numero di (n + 1) bit v tali che x + y + z = u + v. I valori dell'i-esima coppia di bit su sottofondo grigio sono funzioni di x_i, y_i e z_i.* (b) *Un addizionatore senza riporto a 8 bit. Ogni addizionatore completo FA_i ha come input x_i, y_i e z_i e produce un bit di somma u_i e un bit di riporto v_{i+1}. Il bit v_0 è cablato a 0.*

di tempo addizionale. Il trucco è di ridurre il problema di sommare tre numeri al problema di sommare solo due numeri.

Dati tre numeri di n bit $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0\rangle$, $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0\rangle$ e $z = \langle z_{n-1}, z_{n-2}, \ldots, z_0\rangle$, un ***addizionatore senza riporto*** di n bit produce un numero $u = \langle u_{n-1}, u_{n-2}, \ldots, u_0\rangle$ di n bit e un numero $v = \langle v_n, v_{n-1}, \ldots, v_0\rangle$ di $(n + 1)$ bit tali che

$$u + v = x + y + z .$$

Come mostrato nella figura 29.11(a), l'addizionatore senza riporto esegue il calcolo

$$u_i = \text{parità}\ (x_i, y_i, z_i) ,$$

$$v_{i+1} = \text{maggioranza}\ (x_i, y_i, z_i) ,$$

per $i = 0, 1, \ldots, n - 1$. Il bit v_0 è sempre uguale a 0.

L'addizionatore senza riporto di n bit mostrato in figura 29.11(b) consiste di n addizionatori $FA_0, FA_1, \ldots, FA_{n-1}$. Per $i = 0, 1, \ldots, n - 1$, l'addizionatore completo FA_i ha input x_i, y_i e z_i. Il bit di somma di output di FA_i è u_i e il riporto di uscita di FA_i è v_{i+1}. Il bit v_0 è cablato a 0.

Poiché i calcoli di tutti i $2n + 1$ bit di output sono indipendenti, essi possono essere eseguiti in parallelo. Quindi, un addizionatore senza riporto richiede tempo $\Theta(1)$ ed ha dimensione $\Theta(n)$. Per sommare i tre numeri di input, però, bisogna eseguire una somma senza riporto.

impiegando tempo $\Theta(1)$, e poi eseguire una somma con previsione del riporto, impiegando tempo $O(\lg n)$. Sebbene questo metodo asintoticamente non sia migliore del metodo di usare due addizionatori con previsione del riporto, esso, in pratica, è molto più veloce. Inoltre, vedremo nel paragrafo 29.3 che la somma senza riporto è centrale per rendere veloci gli algoritmi per la moltiplicazione.

Esercizi

29.2-1 Siano $a = \langle 01111111\rangle$, $b = \langle 00000001\rangle$ e $n = 8$. Si mostrino i bit di somma e riporto di output degli addizionatori completi quando viene eseguita la somma con propagazione del riporto su queste due sequenze. Si mostrino gli stati del riporto x_0, $x_1, \ldots, x_8$ corrispondenti ad a e b; si associ un'etichetta ad ogni filo del circuito parallelo per il calcolo dei prefissi della figura 29.9 con il valore che assume per gli input x_i e si mostrino gli output $y_0, y_1, \ldots, y_8$ risultanti.

29.2-2 Si provi che l'operatore di stato del riporto $\otimes$ dato nella figura 29.5 è associativo.

29.2-3 Si mostri come costruire un circuito parallelo per il calcolo dei prefissi in tempo $O(\lg n)$ per valori di n che non siano potenze esatte di 2 esaminando il caso per $n = 11$. Si caratterizzino le prestazioni del circuito parallelo per il calcolo dei prefissi costruito sul modello di alberi binari arbitrari.

29.2-4 Si mostri la costruzione della scatola KPG_i a livello di porte logiche. Si assuma che ogni output x_i sia rappresentato da $\langle 00\rangle$ se $x_i = \mathrm{k}$, da $\langle 11\rangle$ se $x_i = \mathrm{g}$ e da $\langle 01\rangle$ o $\langle 10\rangle$ se $x_i = \mathrm{p}$. Si assuma anche che ogni input y_i sia rappresentato da 0 se $y_i = \mathrm{k}$ e da 1 se $y_i = \mathrm{g}$.

29.2-5 Si associ un'etichetta, con l'indicazione della profondità, ad ogni filo nel circuito parallelo per il calcolo dei prefissi della figura 29.9(a). Un ***cammino critico*** in un circuito è il cammino con il più grande numero di elementi combinatori dall'input all'output. Si identifichi il cammino critico nella figura 29.9(a) e si mostri che la sua lunghezza è $O(\lg n)$. Si mostri che qualche nodo ha elementi $\otimes$ che richiedono tempo $\Theta(\lg n)$ a parte. Vi è un nodo i cui elementi $\otimes$ funzionano contemporaneamente?

29.2-6 Si dia un diagramma a blocchi ricorsivo del circuito nella figura 29.12 per qualsiasi numero n di input che sia una potenza esatta di 2. Dedurre sulla base del diagramma a blocchi che in effetti il circuito esegue un calcolo dei prefissi. Si mostri che il circuito ha profondità $\Theta(\lg n)$ e dimensione $\Theta(n \lg n)$.

29.2-7 Qual è il massimo numero di collegamenti di uscita di qualunque filo nell'addizionatore con previsione del riporto? Si mostri che la somma può ancora essere eseguita in tempo $O(\lg n)$ da un circuito di dimensione $\Theta(n)$ anche se si pone la restrizione che le porte abbiano un numero $O(1)$ di collegamenti di uscita.

Figura 29.12 *Un circuito parallelo per il calcolo dei prefissi da usare nell'Esercizio* 29.2-6.

29.2-8 Un ***circuito tally*** ha n input binari e $m = \lceil \lg(n + 1) \rceil$ output. Gli output danno, in binario, il numero di 1 presenti nell'input. Per esempio, se l'input è $\langle 10011110 \rangle$, l'output è $\langle 101 \rangle$, essendoci 5 bit a 1 nell'input. Si descriva un circuito tally di dimensione $\Theta(n)$.

* **29.2-9** Si mostri che la somma su n bit può essere ottenuta con un circuito combinatorio di profondità 4 e dimensione polinomiale in n se si permette che le porte AND e OR abbiano collegamenti di ingresso arbitrariamente elevati. (*Opzionale:* si mostri la stessa cosa ma con profondità 3.)

* **29.2-10** Si supponga che due numeri casuali di n bit siano sommati con un addizionatore con propagazione del riporto, dove ogni bit è indipendentemente 0 o 1 con la stessa probabilità. Si mostri che con probabilità almeno $1 - 1/n$, nessun riporto si propaga per più di $O(\lg n)$ stadi consecutivi. In altre parole, sebbene la profondità dell'addizionatore con propagazione del riporto sia $\Theta(n)$, per due numeri casuali l'output quasi sempre si stabilizza almeno in tempo $O(\lg n)$.

29.3 Circuiti moltiplicatori

Il comune algoritmo di moltiplicazione delle scuole elementari mostrato nella figura 29.13 può calcolare il prodotto $p = \langle p_{2n-1}, p_{2n-2}, \ldots, p_0 \rangle$ su $2n$ bit di due numeri di n bit $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ e $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Si esaminano i bit di b, da b_0 a b_{n-1}. Per ogni bit b_i con valore 1, si somma a al prodotto, ma traslato a sinistra di i posizioni. Per ogni bit b_i con valore 0, si somma 0. Pertanto ponendo $m^{(i)} = a \cdot b_i \cdot 2^i$, si calcola

				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 *Il metodo di moltiplicazione della scuola elementare che viene qui mostrato moltiplica* $a = \langle 1110 \rangle$ *per* $b = \langle 1101 \rangle$ *per ottenere* $p = \langle 10110110 \rangle$. *Si esegue la somma* $\sum_{i=0}^{n-1} m^{(i)}$, *dove* $m^{(i)} = a \cdot b_i \cdot 2^i$. *In questo caso* $n = 4$. *Ogni termine* $m^{(i)}$ *è formato traslando* a *(se* $b_i = 1$*) o 0 (se* $b_i = 0$*) di* i *posizioni a sinistra. I bit che non sono mostrati valgono 0 indipendentemente dai valori di* a *e* b.

$$p = a \cdot b = \sum_{i=0}^{n-1} m^{(i)} .$$

Ogni termine $m^{(i)}$ è chiamato ***prodotto parziale***. Vi sono n prodotti parziali da sommare, con bit che vanno dalla posizione 0 alla posizione $2n - 2$. Il riporto di uscita del bit più significativo è il bit finale in posizione $2n - 1$.

In questo paragrafo, si esamineranno due circuiti per moltiplicare due numeri di n bit. Il moltiplicatore a griglia opera in tempo $\Theta(n)$ ed ha dimensione $\Theta(n^2)$. Anche il moltiplicatore ad albero di Wallace ha dimensione $\Theta(n^2)$ ma opera in tempo $\Theta(\lg n)$. Entrambi i circuiti si basano sull'algoritmo appena visto.

29.3.1 Moltiplicatori a griglia

Un moltiplicatore a griglia concettualmente consiste di tre parti. La prima parte forma i prodotti parziali. La seconda somma i prodotti parziali usando un addizionatore senza riporto. Infine la terza parte somma i due numeri risultanti dalla somma senza riporto usando un addizionatore con propagazione del riporto o con previsione del riporto.

La figura 29.14 mostra un ***moltiplicatore a griglia*** per due numeri di input $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ e $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. I valori a_j passano verticalmente e i valori b_i passano orizzontalmente. Ciascun bit di input si collega in uscita su n porte AND per formare prodotti parziali. Gli addizionatori completi, che sono organizzati come addizionatori senza riporto sommano i prodotti parziali. I bit meno significativi del prodotto finale sono restituiti sulla destra. I bit più significativi vengono calcolati sommando i due numeri di output dell'ultimo addizionatore senza riporto.

Esaminiamo più a fondo la costruzione del moltiplicatore a griglia. Dati due numeri in input $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ e $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, i bit del prodotto parziale sono facili da calcolare. In particolare, per $i, j = 0, 1, \ldots, n - 1$, si ha

$$m^{(i)}_{j+i} = a_j \cdot b_i .$$

Figura 29.14 *Un moltiplicatore a griglia che calcola il prodotto $p = \langle p_{2n-1}, p_{2n-2}, \ldots, p_0 \rangle$ di due numeri di n bit $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ e $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$, mostrato qui per $n = 4$. Ogni porta AND $G_j^{(i)}$ calcola il prodotto parziale $m_j^{(i)}$. Ogni riga di addizionatori completi costituisce un addizionatore senza riporto. Gli n bit meno significativi del prodotto sono $m_0^{(i)}$ e i bit u escono dalla colonna più a destra degli addizionatori completi. Gli n bit più significativi del prodotto sono calcolati sommando i bit di u e v che escono dalla riga di addizionatori completi in basso. Sono mostrati i valori dei bit per gli input $a = \langle 1110 \rangle$ e $b = \langle 1101 \rangle$ e il prodotto $p = \langle 10110110 \rangle$, corrispondenti alle figure 29.13 e 29.15.*

			0	0	0	0		=	0	
			1	1	1	0		=	$m^{(1)}$	
		0	0	0	0			=	$m^{(1)}$	
		0	1	1	1	0		=	$u^{(1)}$	
		0	0	0				=	$v^{(1)}$	
	1	1	1	0				=	$m^{(2)}$	
	1	1	0	1	1	0		=	$u^{(2)}$	
	0	1	0					=	$v^{(2)}$	
1	1	1	0					=	$m^{(3)}$	
1	0	1	0	1	1	0		=	$u^{(3)}$	
1	1	0						=	$v^{(3)}$	
1	0	1	1	0	1	1	0	=	p	

Figura 29.15 *Valutazione della somma dei prodotti parziali attraverso somme senza riporto ripetute. Per esempio, $a = \langle 1110 \rangle$ e $b = \langle 1101 \rangle$. I bit che non sono mostrati valgono 0 indipendentemente dai valori di a e b. Prima si valuta $m^{(0)} + m^{(1)} + 0 = u^{(1)} + v^{(1)}$, poi $u^{(1)} + v^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$, poi $u^{(2)} + v^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$, e infine $p = m^{(0)} + m^{(1)} + m^{(2)} + m^{(3)} = u^{(3)} + v^{(3)}$. Si noti che $p_0 = m_0^{(0)}$ e $p_i = u_i^{(i)}$ per $i = 1, 2, \ldots, n-1$.*

Poiché i prodotti di valori di un bit possono essere calcolati direttamente con una porta AND, tutti i bit dei prodotti parziali (eccetto quelli che valgono 0 che non necessitano di essere esplicitamente calcolati) possono essere prodotti in un passo usando n^2 porte AND.

La figura 29.15 illustra come il moltiplicatore a griglia esegua la somma senza riporto quando si sommano i prodotti parziali nella figura 29.13. Si comincia con la somma senza riporto di $m^{(0)}$, $m^{(1)}$ e 0, ottenendo un numero di $(n+1)$ bit $u^{(1)}$ e un numero di $(n+1)$ bit $v^{(1)}$. (Il numero $v^{(1)}$ ha solo $n+1$ bit, non $n+2$, perché l'$(n+1)$-esimo bit di 0 e di $m^{(0)}$ sono 0.) Pertanto, $m^{(0)} + m^{(1)} = u^{(1)} + v^{(1)}$. Poi viene eseguita la somma senza riporto di $u^{(1)}$, $v^{(1)}$ e $m^{(2)}$, ottenendo un numero di $(n+2)$ bit $u^{(2)}$ e un numero di $(n+2)$ bit $v^{(2)}$. (Ancora, $v^{(2)}$ ha solo $n+2$ bit perché $u_{n+2}^{(1)}$ e $v_{n+2}^{(1)}$ sono 0.) Allora si ha $m^{(0)} + m^{(1)} + m^{(2)} = u^{(2)} + v^{(2)}$. Il moltiplicatore continua con la somma senza riporto di $u^{(i-1)}$, $v^{(i-1)}$ e $m^{(i)}$ per $i = 2, 3, \ldots, n-1$. Il risultato è un numero di $(2n-1)$ bit $u^{(n-1)}$ e un numero di $(2n-1)$ bit $v^{(n-1)}$, dove

$$u^{(n-1)} + v^{(n-1)} = \sum_{i=0}^{n-1} m^{(i)}$$
$$= p \ .$$

In effetti, le somme senza riporto nella figura 29.15 operano su un numero di bit maggiore di quelli strettamente necessari. Si osservi che per $i = 1, 2, \ldots, n-1$ e $j = 0, 1, \ldots, i-1$, si ha $m_j^{(i)} = 0$ a causa della traslazione dei prodotti parziali. Si osservi anche che $v_j^{(i)} = 0$ per $i = 1, 2, \ldots, n-1$ e $j = 0, 1, \ldots, i, i+n, i+n+1, \ldots, 2n-1$. (Si veda l'Esercizio 29.3-1.) Ogni somma senza riporto allora necessita di operare solo su $n-1$ bit.

Si esamini ora la corrispondenza tra il moltiplicatore a griglia e lo schema di somme senza riporto ripetute. Ogni porta AND è etichettata con $G_j^{(i)}$ per qualche i e j negli intervalli $0 \le i \le n-1$ e $0 \le j \le 2n-2$. La porta $G_j^{(i)}$ produce $m_j^{(i)}$, il j-esimo bit dell'i-esimo prodotto parziale. Per $i = 0, 1, \ldots, n-1$, l'i-esima riga delle porte AND calcola gli n bit significativi del prodotto parziale $m^{(i)}$, cioè, $\langle m_{n+i-1}^{(i)}, m_{n+i-2}^{(i)}, \ldots, m_i^{(i)} \rangle$.

Tranne che per gli addizionatori completi nella riga in alto (cioè, per $i = 2, 3, \ldots, n-1$), ogni addizionatore completo $FA_j^{(i)}$ richiede tre bit di input – $m_j^{(i)}$, $u_j^{(i-1)}$ e $v_j^{(i-1)}$ – e produce due

bit di output – $u_j^{(i)}$ e $v_{j+1}^{(i)}$. (Si noti che nella colonna più a sinistra di addizionatori completi si ha $u_{i+n-1}^{(i-1)} = m_{i+n-1}^{(i)}$.) Ogni addizionatore completo $FA_j^{(1)}$ nella riga in alto richiede gli input $m_j^{(0)}$, $m_j^{(1)}$ e 0 e produce i bit $u_j^{(1)}$ e $v_{j+1}^{(1)}$.

Esaminiamo infine l'output del moltiplicatore. Come si è osservato prima, $v_j^{(n-1)}=0$ per $j = 0, 1, \ldots, n-1$. Pertanto $p_j = u_j^{(n-1)}$ per $j = 0,1,\ldots,n-1$. Inoltre, poiché $m_0^{(1)}=0$, si ha $u_0^{(1)} = m_0^{(0)}$ e poiché gli i bit meno significativi di ogni $m^{(i)}$ e $v^{(i+1)}$ valgono 0, si ha $u_j^{(i)} = u_j^{(i-1)}$ per $i = 2, 3, \ldots, n-1$ e $j = 0, 1, \ldots, i-1$. Pertanto $p_0 = m_0^{(0)}$ e, per induzione, $p_i = u_i^{(i)}$ per $i = 1, 2, \ldots, n-1$. I bit del prodotto $\langle p_{2n-1}, p_{2n-2}, \ldots, p_n\rangle$ sono l'uscita di un addizionatore di n bit che somma gli output degli addizionatori completi dell'ultima riga:

$$\langle p_{2n-1}, p_{2n-2}, \ldots, p_n\rangle = \langle u_{2n-2}^{(n-1)}, u_{2n-3}^{(n-1)}, \ldots, u_n^{(n-1)}\rangle + \langle v_{2n-2}^{(n-1)}, v_{2n-3}^{(n-1)}, \ldots, v_n^{(n-1)}\rangle .$$

Analisi

I dati si propagano attraverso un moltiplicatore a griglia da sinistra in alto a destra in basso. Il moltiplicatore richiede tempo $\Theta(n)$ per produrre i bit meno significativi $\langle p_{n-1}, p_{n-2}, \ldots, p_0\rangle$ del prodotto e tempo $\Theta(n)$ perché siano pronti gli input dell'addizionatore. Se l'addizionatore è quello con propagazione del riporto, è richiesto ancora tempo $\Theta(n)$ perché appaiano i bit più significativi $\langle p_{2n-1}, p_{2n-2}, \ldots, p_n\rangle$ del prodotto. Se l'addizionatore è del tipo con previsione del riporto, è necessario solo tempo $\Theta(\lg n)$, ma il tempo complessivo rimane $\Theta(n)$.

Vi sono n^2 porte AND e $n^2 - n$ addizionatori completi nel moltiplicatore a griglia. L'addizionatore che produce i bit più significativi aggiunge solamente altre $\Theta(n)$ porte. Pertanto il moltiplicatore a griglia ha dimensione $\Theta(n^2)$.

29.3.2 Moltiplicatori ad albero di Wallace

Un ***albero di Wallace*** è un circuito che riduce il problema di sommare n numeri di n bit al problema di sommare due numeri di $\Theta(n)$ bit. Esso usa $\lfloor n/3 \rfloor$ addizionatori senza riporto in parallelo per convertire la somma di n numeri nella somma di $\lceil 2n/3 \rceil$ numeri. Quindi costruisce ricorsivamente un albero di Wallace sui $\lceil 2n/3 \rceil$ numeri risultanti. In questo modo, l'insieme di numeri è progressivamente ridotto finché rimangono solo due numeri. Eseguendo molte addizioni senza riporto in parallelo, gli alberi di Wallace permettono di moltiplicare due numeri di n bit in tempo $\Theta(\lg n)$ usando un circuito di dimensione $\Theta(n^2)$.

La figura 29.16 mostra un albero di Wallace[2] che somma 8 prodotti parziali $m^{(0)}, m^{(1)}, \ldots, m^{(7)}$. Il prodotto parziale $m^{(i)}$ consiste di $n+i$ bit. Ogni freccia rappresenta un intero numero, non un solo bit: associato ad ogni linea vi è il numero di bit che rappresenta (si veda l'Esercizio 29.3-3). L'addizionatore con previsione del riporto, in basso, somma un numero di $(2n-1)$ bit a un numero di $2n$ bit per ottenere il prodotto di $2n$ bit.

[2] Come si può vedere dalla figura, un albero di Wallace non è veramente un albero, piuttosto un grafo orientato aciclico. Il nome è storico.

Figura 29.16 *Un albero di Wallace che somma $n = 8$ prodotti parziali $m^{(0)}, m^{(1)}, \ldots, m^{(7)}$. Ogni freccia rappresenta un numero con indicato il numero di bit. L'output a sinistra di ogni addizionatore senza riporto rappresenta i bit di somma e l'output a destra rappresenta i bit di riporto. L'addizionatore con previsione del riporto, in basso, somma un numero di $(2n-1)$ bit a un numero di $2n$ bit per ottenere il prodotto di $2n$ bit.*

Analisi

Il tempo richiesto da un albero di Wallace con n input dipende dalla profondità degli addizionatori senza riporto. Ad ogni livello dell'albero, ogni gruppo di tre numeri è ridotto a due numeri e al più due numeri sono passati al livello successivo (come nel caso di $m^{(6)}$ e $m^{(7)}$ al primo livello). Pertanto la massima profondità $D(n)$ di un addizionatore senza riporto in un albero di Wallace con n input è data dalla ricorrenza

$$D(n) = \begin{cases} 0 & \text{se } n \le 2 , \\ 1 & \text{se } n = 3 , \\ D(\lceil 2n/3 \rceil) + 1 & \text{se } n \ge 4 , \end{cases}$$

che ha soluzione $D(n) = \Theta(\lg n)$ per il caso 2 del teorema principale (Teorema 4.1). Ogni addizionatore senza riporto richiede tempo $\Theta(1)$. Tutti gli n prodotti parziali possono essere eseguiti in parallelo in tempo $\Theta(1)$. (Gli $i-1$ bit meno significativi di $m^{(i)}$, per $i = 1, 2, \ldots, n-1$, sono cablati a 0.) L'addizionatore con previsione del riporto richiede tempo $O(\lg n)$. Pertanto, la moltiplicazione di due numeri di n bit richiede complessivamente tempo $\Theta(\lg n)$.

Un moltiplicatore ad albero di Wallace per due numeri di n bit ha dimensione $\Theta(n^2)$, come si può vedere nel modo seguente. Si limita dapprima la dimensione del circuito degli addizionatori senza riporto. Un limite inferiore di $\Omega(n^2)$ è facile da ottenere, poiché vi sono

$\lfloor n/3 \rfloor$ addizionatori senza riporto a profondità 1 e ognuno consiste di almeno n addizionatori completi. Per ottenere il limite superiore di $O(n^2)$, si osservi che, poiché il prodotto finale ha $2n$ bit, ogni addizionatore senza riporto nell'albero di Wallace contiene al più $2n$ addizionatori completi. È necessario mostrare che vi sono complessivamente $O(n)$ addizionatori senza riporto. Sia $C(n)$ il numero totale di addizionatori senza riporto in un albero di Wallace con n numeri di input. Si ha la ricorrenza

$$C(n) = \begin{cases} 1 & \text{se } n = 3 , \\ C(\lceil 2n/3 \rceil) + \lfloor n/3 \rfloor & \text{se } n \geq 4 , \end{cases}$$

che ha soluzione $C(n) = \Theta(n)$ per il caso 3 del teorema principale. Si ottiene così un limite asintoticamente stretto di $\Theta(n^2)$ per gli addizionatori senza riporto di un moltiplicatore ad albero di Wallace. Il circuito per generare gli n prodotti parziali ha dimensione $\Theta(n^2)$ e l'addizionatore con previsione del riporto alla fine ha dimensione $\Theta(n)$. Pertanto la dimensione dell'intero moltiplicatore è $\Theta(n^2)$.

Sebbene il moltiplicatore ad albero di Wallace sia asintoticamente più veloce del moltiplicatore a griglia ed abbia la stessa dimensione asintotica, quando viene realizzato la sua struttura non è regolare come quella di un moltiplicatore a griglia né così "densa" (nel senso che viene sprecato poco spazio tra gli elementi del circuito). In pratica è spesso usato un compromesso tra i due progetti. L'idea è di usare due griglie in parallelo, una che sommi metà dei prodotti parziali e una che sommi l'altra metà. Il ritardo di propagazione è solo la metà di quello che si verifica con una singola griglia che somma tutti gli n prodotti parziali. Due ulteriori addizioni senza riporto riducono le 4 uscite delle griglie a 2 numeri e l'addizionatore con previsione del riporto somma i 2 numeri per ottenere il prodotto. Il ritardo di propagazione totale è un po' più della metà di quello di un moltiplicatore a griglia piena, più un ulteriore termine $O(\lg n)$.

Esercizi

29.3-1 Si dimostri che in un moltiplicatore a griglia si ha $v_j^{(i)} = 0$ per $i = 1, 2, \ldots, n-1$ e $j = 0, 1, \ldots, i, i+n, i+n+1, \ldots, 2n-1$.

29.3-2 Si mostri che nel moltiplicatore a griglia della figura 29.14, tutti gli addizionatori completi tranne uno nella linea in alto non sono necessari. Ci sarà la necessità di rivedere la connessione dei fili.

29.3-3 Si supponga che un addizionatore senza riporto prenda gli input x, y, z e produca gli output s e c, con n_x, n_y, n_z, n_s e n_c bit rispettivamente. Si supponga anche, senza perdita di generalità, che $n_y \leq n_z \leq n_x$. Si mostri che $n_s = n_z$ e che

$$n_c = \begin{cases} n_z & \text{se } n_y \leq n_z , \\ n_z + 1 & \text{se } n_y = n_z . \end{cases}$$

29.3-4 Si mostri che la moltiplicazione può ancora essere eseguita in tempo $O(\lg n)$ con dimensione $O(n^2)$ anche se si pone la restrizione che le porte debbano avere $O(1)$ collegamenti di uscita.

29.3-5 Si descriva un circuito efficiente per calcolare il quoziente della divisione per 3 di un numero binario. (*Suggerimento:* si noti che in binario $0.010101\ldots = 0.01 \times 1.01 \times 1.0001 \times \cdots$.)

29.3-6 Un ***traslatore ciclico***, è un circuito che ha due input $x = \langle x_{n-1}, x_{n-2}, \ldots, x_0 \rangle$ e $s = \langle s_{m-1}, s_{m-2}, \ldots, s_0 \rangle$, dove $m = \lceil \lg n \rceil$. Il suo output $y = \langle y_{n-1}, y_{n-2}, \ldots, y_0 \rangle$ è specificato da $y_i = x_{i+s \bmod n}$, per $i = 0, 1, \ldots, n-1$. Cioè il traslatore ruota i bit di x di s posizioni. Si descriva un traslatore ciclico efficiente. In termini di moltiplicazione modulare, quale funzione realizza un traslatore ciclico?

29.4 Circuiti sequenziali

Gli elementi di un circuito combinatorio sono usati solo una volta durante un calcolo. Introducendo elementi di memoria temporizzati nel circuito, si possono riutilizzarne gli elementi combinatori. Dato che possono usare l'hardware più di una volta, i circuiti sequenziali possono spesso essere molto più piccoli dei circuiti combinatori che calcolano la stessa funzione.

In questo paragrafo si studiano i circuiti sequenziali per l'esecuzione di somma e moltiplicazione. Si comincia con un circuito sequenziale di dimensione $\Theta(1)$, chiamato addizionatore seriale, che può sommare due numeri di n bit in tempo $\Theta(n)$. Quindi si esaminano i moltiplicatori a catena. Si presenta un moltiplicatore a catena di dimensione $\Theta(n)$ che può moltiplicare due numeri di n bit in tempo $\Theta(n)$.

29.4.1 Addizione seriale

Introduciamo la nozione di circuito sequenziale ritornando al problema di sommare due numeri di n bit. La figura 29.17 mostra come si possa usare un singolo addizionatore completo per produrre su $n+1$ bit la somma $s = \langle s_n, s_{n-1}, \ldots, s_0 \rangle$ di due numeri di n bit $a = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ e $b = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Dall'esterno viene presentata una coppia alla volta dei bit in input: prima a_0 e b_0, poi a_1 e b_1 e così via. Sebbene il bit di riporto di uscita di una posizione sia il bit di riporto di ingresso della posizione successiva, non si può semplicemente fornire l'output di un addizionatore completo direttamente in un input. Vi è un problema di temporizzazione: il riporto di ingresso c_i che entra nell'addizionatore completo deve corrispondere agli input a_i e b_i appropriati. A meno che questi input arrivino esattamente nello stesso istante del riporto da fornire in input, l'output può essere scorretto.

Come mostrato nella figura 29.17, la soluzione consiste nell'usare un ***circuito temporizzato***, o ***circuito sequenziale***, formato da un circuito combinatorio e uno o più ***registri*** (elementi di memoria temporizzati). Il circuito combinatorio riceve gli input dall'esterno o dalle uscite dei registri. Fornisce output all'esterno e all'input dei registri. Come in un circuito combinatorio, non è permesso che il circuito combinatorio di un circuito sequenziale contenga dei cicli.

Ogni registro in un circuito sequenziale è controllato da un segnale periodico, detto ***clock***. Ad ogni ***impulso*** di clock, il registro carica il suo input e lo memorizza. Il periodo di tempo tra due impulsi di clock successivi si dice ***ciclo di clock***. In un circuito ***temporizzato globalmente***, ogni registro riceve lo stesso segnale di clock.

Figura 29.17 *Il funzionamento di un addizionatore seriale. Durante l'i-esimo ciclo di clock, per $i = 0, 1, \ldots, n$, l'addizionatore completo FA prende i bit di input a_i e b_i dall'esterno e un bit di riporto dal registro. L'addizionatore completo quindi calcola il bit di somma s_i, che viene fornito all'esterno e il bit di riporto c_{i+1}, che è memorizzato nel registro che dovrà essere usato durante il prossimo ciclo di clock. Il registro è inizializzato con $c_0 = 0$. (a) - (e) Lo stato del circuito in ognuno dei cinque cicli di clock durante la somma di $a = \langle 1011 \rangle$ e $b = \langle 1001 \rangle$ per produrre $s = \langle 10100 \rangle$.*

Esaminiamo ora in maggior dettaglio il funzionamento di un registro. Ogni impulso di clock è da considerarsi istantaneo. Ad un dato impulso, un registro legge il valore di input che gli si presenta *in quel momento* e lo memorizza. Questo valore memorizzato appare quindi sull'output del registro dove può essere usato per calcolare i valori che saranno messi negli altri registri al prossimo impulso. In altre parole, il valore sull'input di un registro durante un ciclo di clock appare sull'output del registro durante il ciclo di clock successivo.

Consideriamo ora il circuito nella figura 29.17, chiamato ***addizionatore seriale***. Perché l'output di un addizionatore completo sia corretto, si richiede che il ciclo di clock duri almeno quanto il ritardo di propagazione dell'addizionatore completo in modo che il circuito combinatorio abbia la possibilità di stabilizzarsi tra due impulsi. Durante il ciclo di clock 0, mostrato nella figura 29.17(a), i bit di input a_0 e b_0 giungono dall'esterno a due degli input dell'addizionatore completo. Si assume che il registro sia inizializzato a 0; il bit di riporto di ingresso iniziale, che è l'output del registro, è pertanto $c_0 = 0$. Quindi, in questo ciclo di clock, escono dall'addizionatore completo i bit di somma s_0 e il riporto di uscita c_1. Il bit di somma va all'esterno dove presumibilmente sarà salvato come parte dell'intera somma s. Il filo del riporto di uscita dell'addizionatore viene collegato al registro, così che c_1 viene letto nel registro al prossimo impulso. All'inizio del ciclo di clock 1, mostrato nella figura 29.17(b), a_1 e b_1 giungono dall'esterno all'addizionatore completo che, leggendo c_1 dal registro, produce gli output s_1 e c_2. Il bit di somma s_1 va all'esterno e c_2 va al registro. Il ciclo continua fino al ciclo di clock n, mostrato nella figura 29.17(e), in cui il registro contiene c_n. Dall'esterno giungono $a_n = b_n = 0$, così da avere $s_n = c_n$.

Analisi

Per determinare il tempo totale t richiesto da un circuito temporizzato globalmente, bisogna conoscere il numero p di cicli di clock e la durata d di ogni ciclo di clock: $t = pd$. Il ciclo di clock d deve essere sufficientemente lungo perché il circuito combinatorio si stabilizzi tra due impulsi. Sebbene per qualche input si possa stabilizzare prima, per far funzionare correttamente il circuito per tutti gli input, d deve essere almeno proporzionale alla profondità del circuito combinatorio.

Quanto tempo richiede la somma seriale di due numeri di n bit? Ogni ciclo di clock richiede tempo $\Theta(1)$ perché la profondità dell'addizionatore completo è $\Theta(1)$. Poiché sono richiesti $n + 1$ impulsi per produrre tutti gli output, il tempo totale per eseguire la somma seriale è $(n + 1)\,\Theta(1) = \Theta(n)$.

La dimensione dell'addizionatore seriale (il numero di elementi combinatori più il numero di registri) è $\Theta(1)$.

Confronto tra addizione con propagazione del riporto e addizione seriale

Si osservi che un addizionatore con propagazione del riporto è come un addizionatore seriale replicato con i registri sostituiti dalle connessioni dirette tra gli elementi combinatori. Cioè, l'addizionatore con propagazione del riporto corrisponde a uno "sviluppo" nello spazio del comportamento dell'addizionatore seriale. L'i-esimo addizionatore completo nell'addizionatore con propagazione del riporto realizza l'i-esimo ciclo di clock dell'addizionatore seriale.

In generale, si può sostituire ogni circuito sequenziale con un circuito combinatorio equivalente che abbia lo stesso ritardo asintotico di tempo se si conosce in anticipo per quanti periodi di tempo il circuito sequenziale funzionerà. Naturalmente vi è da fare un bilancio. Il circuito sequenziale usa un numero inferiore di elementi nel circuito (un fattore di $\Theta(n)$ in meno per l'addizionatore seriale rispetto all'addizionatore con propagazione del riporto), ma il circuito combinatorio ha il vantaggio di un minor controllo del circuito - non c'è bisogno di clock o ci circuiti esterni sincronizzati per presentare i bit di input e memorizzare i bit di somma. Inoltre, sebbene i circuiti abbiano lo stesso ritardo di tempo asintotico, tipicamente il circuito combinatorio è, in pratica, leggermente più veloce. La maggior velocità è possibile perché il circuito combinatorio non deve aspettare che i valori si stabilizzino durante ogni ciclo di clock. Se tutti gli input si stabilizzano insieme, i valori attraversano il circuito alla massima velocità possibile, senza aspettare il clock.

29.4.2 Moltiplicatori a catena

Il moltiplicatore parallelo del paragrafo 29.3 richiede dimensione $\Theta(n^2)$ per moltiplicare due numeri di n bit. Presentiamo ora due moltiplicatori che sono griglie lineari (o catene), piuttosto che bidimensionali, di elementi combinatori. Come il moltiplicatore a griglia, il più veloce di questi due moltiplicatori a catena impiega tempo $\Theta(n)$.

Il moltiplicatore a catena realizza l'***algoritmo del contadino russo*** (così detto perché gli occidentali che visitarono la Russia nel diciannovesimo secolo trovarono che in quel paese l'algoritmo era molto usato) illustrato nella figura 29.18(a). Dati due numeri di input a e b, si formano due colonne di numeri con a e b in testa. In ogni riga, il numero della colonna di a è metà del numero sulla riga precedente della stessa colonna, ignorando la parte frazionaria. Il numero nella colonna di b è il doppio del numero nella riga precedente della stessa colonna. L'ultima riga è quella con un 1 nella colonna di a. Si cercano tutti i numeri dispari della colonna di a e si sommano i numeri corrispondenti nella colonna di b. Questa somma è il prodotto $a \cdot b$.

a	b
19	29
9	58
4	116
2	232
1	464
	551

(a)

a	b
10011	11101
1001	11101
100	1110100
10	11101000
1	111010000
	1000100111

(b)

Figura 29.18 *Moltiplicazione di 19 per 29 con l'algoritmo del contadino russo. In ogni riga il numero nella colonna di a è metà del numero nella precedente riga con la parte frazionaria ignorata, mentre i numeri nella colonna b raddoppiano di riga in riga. Si sommano i numeri nella colonna di b di tutte le righe che abbiano un numero dispari nella colonna di a, evidenziati in grigio. Questa somma è il prodotto desiderato.* (a) *I numeri espressi in decimale.* (b) *Gli stessi numeri espressi in binario.*

Sebbene a prima vista l'algoritmo del contadino russo possa sembrare straordinario, la figura 29.18(b) mostra che in effetti è solo una realizzazione del metodo di moltiplicazione delle scuole elementari con un sistema numerico binario, ma con i numeri espressi in decimale invece che in binario. Le righe in cui il numero nella colonna di a è dispari contribuiscono al prodotto con un termine b moltiplicato per l'appropriata potenza di 2.

Realizzazione lenta

La figura 29.19(a) mostra un modo di realizzare l'algoritmo di moltiplicazione del contadino russo per due numeri di n bit. Si usa un circuito sequenziale che consiste di una catena di $2n$ celle. Ogni cella contiene tre registri. Un registro contiene un bit del numero a, uno contiene un bit del numero b e uno contiene un bit del prodotto p. Per denotare i valori delle celle prima di ogni passo dell'algoritmo, si scrive all'apice l'indice del passo. Per esempio, il valore del bit a_i prima del j-esimo passo è $a_i^{(j)}$ e si definisce $a(j) = \langle a_{2n-1}^{(j)}, a_{2n-2}^{(j)}, \ldots, a_0^{(j)} \rangle$.

L'algoritmo esegue una sequenza di n passi, numerati 0, 1, ..., $n-1$, dove ogni passo richiede un ciclo di clock. L'algoritmo ha come invariante che, prima del j-esimo passo, vale

$$a^{(j)} \cdot b^{(j)} + p^{(j)} = a \cdot b \tag{29.6}$$

(si veda l'Esercizio 29.4-2). Inizialmente, $a^{(0)} = a$, $b^{(0)} = b$ e $p^{(0)} = 0$. Il passo j-esimo consiste dei seguenti calcoli.

1. Se $a^{(j)}$ è dispari (cioè, se $a_0^{(j)} = 1$), allora si somma b a p: $p^{(j+1)} \leftarrow b^{(j)} + p^{(j)}$. (La somma è eseguita da un addizionatore con propagazione del riporto lungo la catena: i bit di riporto scorrono da destra a sinistra.) Se $a^{(j)}$ è pari, allora il riporto di p va al passo successivo: $p^{(j+1)} \leftarrow p^{(j)}$.
2. Si trasla a di una posizione a destra:

$$a_i^{(j+1)} \leftarrow \begin{cases} a_{i+1}^{(j)} & \text{se } 0 \le i \le 2n-2 \ , \\ 0 & \text{se } i = 2n-1 \ . \end{cases}$$

Figura 29.19 *Due realizzazioni a catena dell'algoritmo del contadino russo per la moltiplicazione di $a = 19 = \langle 10011 \rangle$ e $b = 29 = \langle 11101 \rangle$, con $n = 5$. È mostrata la situazione all'inizio di ogni passo j, con i bit significativi di $a^{(j)}$ e $b^{(j)}$ in grigio.* (a) *Una realizzazione lenta che impiega tempo $\Theta(n^2)$. Poiché $a^{(5)} = 0$, si ha $p^{(5)} = a \cdot b$. Vi sono n passi e ad ogni passo si usa una somma con propagazione del riporto. Il ciclo di clock è quindi proporzionale alla lunghezza della catena, $\Theta(n)$, portando a un tempo complessivo $\Theta(n^2)$.* (b) *Una realizzazione veloce che impiega tempo $\Theta(n)$ perché ogni passo usa una somma senza riporto piuttosto che una somma con propagazione del riporto, richiedendo così solo tempo $\Theta(1)$. Il numero totale di passi è $2n - 1 = 9$; dopo l'ultimo passo mostrato, le ulteriori somme senza riporto di u e v portano a $u^{(9)} = a \cdot b$.*

3. Si trasla b di una posizione a sinistra:

$$b_i^{(j+1)} \leftarrow \begin{cases} b_{i-1}^{(j)} & \text{se } 1 \le i \le 2n-1 \ . \\ 0 & \text{se } i = 0 \ . \end{cases}$$

Dopo l'esecuzione di n passi, tutti i bit di a sono stati traslati; pertanto $a^{(n)} = 0$. L'invariante (29.6) implica quindi che $p^{(n)} = a \cdot b$.

Si analizza ora l'algoritmo. Sono richiesti n passi, assumendo che l'informazione di controllo sia passata ad ogni cella simultaneamente. Ogni passo impiega tempo $\Theta(n)$ nel caso peggiore perché la profondità dell'addizionatore con propagazione del riporto è $\Theta(n)$, così la durata del ciclo di clock deve essere almeno $\Theta(n)$. Ogni traslazione richiede solo tempo $\Theta(1)$. Quindi, complessivamente, l'algoritmo richiede tempo $\Theta(n^2)$. Poiché ogni cella ha dimensione costante, l'intera catena ha dimensione $\Theta(n)$.

Realizzazione veloce

Usando una somma senza riporto invece di quella con propagazione del riporto, si può diminuire il tempo di ogni passo portandolo a $\Theta(1)$, abbassando il tempo complessivo a $\Theta(n)$. Come mostra la figura 29.19(b), anche in questo caso ogni cella contiene un bit del numero a e un bit del numero b. Ogni cella contiene anche altri due bit di u e v che sono le uscite di un addizionatore senza riporto. Usando una rappresentazione senza riporto per accumulare il prodotto, si mantiene l'invariante che prima del j-esimo passo vale

$$a^{(j)} \cdot b^{(j)} + u^{(j)} + v^{(j)} = a \cdot b \qquad (29.7)$$

(si veda ancora l'Esercizio 29.4-2). Ad ogni passo si traslano a e b come nella realizzazione lenta, così che si possano combinare le equazioni (29.6) e (29.7) per ottenere $u^{(j)} + v^{(j)} = p^{(j)}$. Pertanto, i bit di u e v contengono le stesse informazioni dei bit di p nella realizzazione lenta.

Il j-esimo passo della realizzazione veloce esegue la somma senza riporto di u e v, dove gli operandi dipendono dal fatto che a sia pari o dispari. Se $a_0^{(j)} = 1$, si calcola

$$u_i^{(j+1)} \leftarrow \text{parità}\,(b_i^{(j)}, u_i^{(j)}, v_i^{(j)}) \qquad \text{per } i = 0, 1, \ldots, 2n-1$$

e

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{maggioranza}(b_{i-1}^{(j)}, u_{i-1}^{(j)}, v_{i-1}^{(j)}) & \text{se } 1 \le i \le 2n-1 \ . \\ 0 & \text{se } i = 0 \ . \end{cases}$$

Altrimenti $a_0^{(j)} = 0$ e si calcola

$$u_i^{(j+1)} = \text{parità}\,(0, u_i^{(j)}, v_i^{(j)}) \qquad \text{per } i = 0, 1, \ldots, 2n-1$$

e

$$v_i^{(j+1)} \leftarrow \begin{cases} \text{maggioranza}(0, u_{i-1}^{(j)}, v_{i-1}^{(j)}) & \text{se } 1 \le i \le 2n-1 \ . \\ 0 & \text{se } i = 0 \ . \end{cases}$$

Dopo l'aggiornamento di u e v, il j-esimo passo trasla a a destra e b a sinistra come nella realizzazione lenta.

La realizzazione veloce esegue un totale di $2n-1$ passi. Per $j \ge n$, si ha $a^{(j)} = 0$ e l'invariante (29.7) implica perciò $u^{(n)} + v^{(n)} = a \cdot b$. Quando $a^{(j)} = 0$, tutti i passi successivi servono solo a

eseguire la somma senza riporto di u e v. L'Esercizio 29.4-3 richiede di dimostrare che $v^{(2n-1)} = 0$ in modo che $u^{(2n-1)} = a \cdot b$.

Il tempo totale richiesto nel caso peggiore è $\Theta(n)$, poiché ognuno dei $2n-1$ passi richiede tempo $\Theta(1)$. Poiché ogni cella ha dimensione costante, la dimensione totale rimane $\Theta(n)$.

Esercizi

29.4-1 Siano $a = \langle 101101 \rangle$, $b = \langle 011110 \rangle$ e $n = 6$. Mostrare il funzionamento dell'algoritmo del contadino russo sia in binario che in decimale per gli input a e b.

29.4-2 Si dimostrino le invarianti (29.6) e (29.7) per i moltiplicatori a catena.

29.4-3 Si dimostri che in un moltiplicatore a catena veloce $v^{(2n-1)} = 0$.

29.4-4 Si descriva come il moltiplicatore a griglia del paragrafo 29.3.1 rappresenti una "espansione" del calcolo di un moltiplicatore a catena veloce.

29.4-5 Si consideri un flusso di dati $\langle x_1, x_2, \ldots \rangle$ che arriva ad un circuito sequenziale al ritmo di un valore per impulso. Per un dato valore n, il circuito deve calcolare il valore

$$y_t = \max_{t-n+1 \le i \le t} x_i$$

per $t = n, n+1, \ldots$. Cioè, y_t è il massimo degli n valori più recenti ricevuti dal circuito. Si descriva un circuito di dimensione $O(n)$ che ad ogni impulso di clock riceve il valore x_t e calcola il valore di uscita y_t in tempo $O(1)$. Il circuito può usare registri ed elementi combinatori che calcolino il massimo tra due input.

* *29.4-6* Ripetere l'Esercizio 29.4-5 usando solo $O(\lg n)$ elementi per calcolare il massimo.

Problemi

29-1 Circuiti di divisione

Si può costruire un circuito di divisione con circuiti sottrattori e moltiplicatori con una tecnica chiamata ***iterazione di Newton***. Si considererà il problema correlato di calcolare il reciproco, poiché si può ottenere un circuito di divisione facendo una moltiplicazione in più.

L'idea è di calcolare una sequenza $y_0, y_1, y_2, \ldots$ di approssimazioni del reciproco di un numero x usando la formula

$$y_{i+1} \leftarrow 2y_i - xy_i^2 \ .$$

Si assuma che x sia dato come una frazione binaria di n bit presa nell'intervallo $1/2 \le x \le 1$. Poiché il reciproco può essere un numero frazionario che si ripete all'infinito, ci si concentrerà sul calcolo di un'approssimazione su n bit esatta fino al bit meno significativo.

a. Si supponga che $|y_i - 1/x| \le \varepsilon$ per qualche costante $\varepsilon > 0$. Si provi che $|y_{i+1} - 1/x| \le \varepsilon^2$.

b. Si dia un'approssimazione iniziale y_0 tale che y_k soddisfi $|y_k - 1/x| \le 2^{-2^k}$ per ogni $k \ge 0$. Quanto deve essere grande k affinché l'approssimazione y_k sia precisa fino al bit meno significativo?

c. Si descriva un circuito combinatorio che, dato un input x di n bit, calcoli un'approssimazione su n bit di $1/x$ in tempo $O(\lg^2 n)$. Qual è la dimensione del circuito? (*Suggerimento:* con un po' di astuzia si può migliorare il limite $\Theta(n^2 \lg n)$.)

29-2 *Formule Booleane per funzioni simmetriche*

Una funzione con n input $f(x_1, x_2, \ldots, x_n)$ è ***simmetrica*** se

$$f(x_1, x_2, \ldots, x_n) = f(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$$

per qualunque permutazione π di $\{1, 2, \ldots, n\}$. In questo problema si mostrerà che vi è una formula booleana che rappresenta f la cui dimensione è polinomiale in n. (In questo contesto, una formula booleana è una stringa che può contenere le variabili $x_1, x_2, \ldots, x_n$, le parentesi e gli operatori booleani $\vee$, $\wedge$, $\neg$.) L'approccio sarà di convertire un circuito booleano di profondità logaritmica in una formula booleana equivalente di dimensione polinomiale. Si assumerà che tutti i circuiti siano costruiti con porte AND e OR a due input e porte NOT.

a. Si comincia considerando una semplice funzione simmetrica. La ***funzione di maggioranza*** generalizzata di n input booleani è definita da

$$\text{maggioranza}_n(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{se } x_1 + x_2 + \ldots + x_n > n/2 , \\ 0 & \text{altrimenti} . \end{cases}$$

Si descriva un circuito combinatorio di profondità $O(\lg n)$ per maggioranza$_n$. (*Suggerimento:* costruire un albero di addizionatori.)

b. Si supponga che f sia una funzione booleana arbitraria nelle n variabili booleane $x_1, x_2, \ldots, x_n$. Si supponga inoltre che vi sia un circuito C di profondità d che calcola f. Si mostri come costruire da C una formula booleana per f di lunghezza $O(2^d)$. Si concluda che vi è una formula di dimensione polinomiale per maggioranza$_n$.

c. Si deduca che qualunque funzione booleana simmetrica $f(x_1, x_2, \ldots, x_n)$ può essere espressa come funzione di $\sum_{i=1}^{n} x_i$.

d. Si deduca che qualunque funzione simmetrica su n input booleani può essere calcolata da un circuito combinatorio di profondità $O(\lg n)$.

e. Si deduca che qualunque funzione booleana simmetrica di n variabili booleane può essere rappresentata da una formula booleana la cui lunghezza è polinomiale rispetto a n.

Note al capitolo

Molti libri sull'aritmetica dei calcolatori si concentrano maggiormente sulla realizzazione pratica dei circuiti piuttosto che sulla teoria degli algoritmi. Savage [173] è uno dei pochi che studia gli aspetti algoritmici dell'argomento. I libri sull'aritmetica dei calcolatori più orientati all'hardware di Cavanagh [39] e di Hwang [108] sono particolarmente interessanti. Buoni libri sul progetto logico combinatorio e sequenziale includono Hill e Peterson [96] e, con accenni alla teoria dei linguaggi formali, Kohavi [126].

Aiken e Hopper [7] hanno ricostruito la storia degli algoritmi aritmetici. La somma con propagazione del riporto è vecchia almeno quanto l'abaco, che è utilizzato da più di 5000 anni. Il primo calcolatore meccanico che impiegò la somma con propagazione del riporto fu escogitato da B. Pascal nel 1642. Una macchina di calcolo per la moltiplicazione basata su somme ripetute fu concepita da S. Morland nel 1666 e indipendentemente da G. W. Leibnitz nel 1671. L'algoritmo del contadino russo per la moltiplicazione è apparentemente molto più vecchio del suo uso in Russia nel diciannovesimo secolo. Secondo Knuth[122], fu usato dai matematici egiziani fin dal 1800 a.C.

Gli stati (eliminato, generato e propagato) di una catena di riporti, furono impiegati in un calcolatore a relé costruito ad Harward durante la metà del 1940 [180]. Una delle prime realizzazioni della somma con previsione del riporto fu descritta da Weinberger e Smith [199], ma il loro metodo di previsione richiede porte di notevoli dimensioni. Ofman [152] provò che numeri di n bit potrebbero essere sommati in tempo $O(\lg n)$ usando la somma con previsione del riporto con porte di dimensione costante.

L'idea di usare la somma senza riporto per accelerare la moltiplicazione è dovuta a Estrin, Gilchrist e Pomerene [64]. Atrubin [13] descrive un moltiplicatore a catena di lunghezza infinita che può essere usato per moltiplicare numeri binari di lunghezza arbitraria. Il moltiplicatore produce gli n bit del prodotto immediatamente, appena riceve gli n bit degli input. Il moltiplicatore ad albero di Wallace è attribuito a Wallace [197], ma l'idea fu scoperta indipendentemente anche da Ofman [152].

Gli algoritmi di divisione risalgono a I. Newton che attorno al 1665 inventò quella che è poi diventata nota come iterazione di Newton. Il Problema 29-1 usa l'iterazione di Newton per costruire un circuito di divisione con profondità $\Theta(\lg^2 n)$. Questo metodo fu poi migliorato da Beame, Cook e Hoover [19] che mostrarono che la divisione su n bit può, di fatto, essere eseguita con profondità $\Theta(\lg n)$.

Algoritmi per calcolatori paralleli

30

Con il proliferare dei calcolatori ad elaborazione parallela è cresciuto l'interesse negli ***algoritmi paralleli***: algoritmi che eseguono più di un'operazione alla volta. Lo studio degli algoritmi paralleli è diventato un'area di ricerca a sé stante. Infatti, sono stati sviluppati algoritmi paralleli per la maggior parte dei problemi che in questo testo sono stati risolti usando comuni algoritmi sequenziali. In questo capitolo, si descriveranno alcuni semplici algoritmi paralleli che illustrano le questioni e le tecniche fondamentali.

Per studiare gli algoritmi paralleli si deve scegliere un modello appropriato per il calcolo parallelo. La macchina ad accesso diretto, o RAM, che è stata usata nella maggior parte di questo libro, è naturalmente seriale e non parallela. I modelli paralleli considerati – reti di ordinamento (Capitolo 28) e circuiti (Capitolo 29) – sono troppo restrittivi per studiare, per esempio, algoritmi sulle strutture di dati.

Gli algoritmo paralleli sono presentati in questo capitolo in termini di un noto modello teorico: la macchina parallela ad accesso diretto, o PRAM (dall'inglese "parallel random-access machine", si pronuncia "pi-ram"). Molti algoritmi paralleli per array, liste, alberi e grafi possono essere facilmente descritti nel modello PRAM. Sebbene la PRAM ignori molti importanti aspetti delle macchine parallele reali, gli attributi essenziali degli algoritmi paralleli tendono a superare i modelli per i quali sono stati progettati. Se un algoritmo PRAM è migliore di un altro algoritmo PRAM, le prestazioni relative probabilmente non cambiano in modo sostanziale qualora entrambi gli algoritmi siano adattati per essere eseguiti su un calcolatore parallelo reale.

Il modello PRAM

La figura 30.1 mostra l'architettura di base della ***macchina parallela ad accesso diretto*** (***PRAM***). Vi sono p processori seriali $P_0, P_1, \ldots, P_{p-1}$ che hanno come memoria una memoria globale condivisa. Tutti i processori possono leggere o scrivere nella memoria globale "in parallelo" (contemporaneamente). I processori possono eseguire in parallelo anche varie operazioni logiche e aritmetiche.

L'ipotesi chiave riguardo alle prestazioni nel modello PRAM è che il tempo di esecuzione può essere misurato come il numero di accessi paralleli alla memoria eseguiti da un algoritmo. Questa ipotesi è una generalizzazione corretta del comune modello RAM in cui il numero di accessi alla memoria è asintoticamente altrettanto buono quanto ogni altra misura del tempo di esecuzione. Questa semplice ipotesi sarà utile nella rassegna sugli algoritmi paralleli, anche se i calcolatori paralleli reali non possono eseguire accessi paralleli alla memoria

Figura 30.1 *L'architettura di base della PRAM. Vi sono p processori $P_0, P_1, \ldots, P_{p-1}$ connessi a una memoria condivisa. Ogni processore può accedere ad una qualunque parola di memoria condivisa nell'unità di tempo.*

globale in un'unità di tempo: il tempo di un accesso alla memoria cresce col numero di processori presenti nel calcolatore parallelo.

Tuttavia, per algoritmi paralleli che accedono ai dati in modo arbitrario, l'ipotesi sulle operazioni in memoria in un'unità di tempo può essere giustificata. Le macchine parallele reali hanno tipicamente una rete di comunicazione che consente l'astrazione di una memoria globale. L'accesso ai dati attraverso la rete è un'operazione relativamente lenta in confronto alle operazioni aritmetiche. Pertanto, contare il numero di accessi paralleli alla memoria eseguiti da due algoritmi paralleli porta in realtà a una stima alquanto accurata delle loro prestazioni relative. Il modo principale in cui le macchine parallele violano l'astrazione sull'unità di tempo del modello PRAM è che alcune configurazioni di accesso alla memoria sono più veloci di altre. Come prima approssimazione, però l'ipotesi sull'unità di tempo nel modello PRAM è abbastanza ragionevole.

Il tempo di esecuzione di un algoritmo parallelo dipende sia dal numero di processori che eseguono l'algoritmo che dalla dimensione dell'input del problema. Generalmente, però, quando si analizzano gli algoritmi PRAM si devono discutere sia il tempo che il numero di processori; ciò contrasta con gli algoritmi seriali, per l'analisi dei quali si è concentrata l'attenzione principalmente sul tempo. Tipicamente, vi è una contrapposizione tra il numero di processori usati da un algoritmo e il suo tempo di esecuzione. Il paragrafo 30.3 discute queste contrapposizioni.

Confronto tra accessi concorrenti ed esclusivi alla memoria

Un algoritmo a ***lettura-concorrente*** è un algoritmo PRAM durante l'esecuzione del quale più di un processore può leggere della stessa locazione di memoria condivisa nello stesso istante. Un algoritmo a ***lettura-esclusiva*** è un algoritmo PRAM per cui non ci sono mai due processori che leggono la stessa locazione di memoria nello stesso istante. Si fa una distinzione simile se più processori possono scrivere nella stessa locazione di memoria nello stesso istante, dividendo gli algoritmi PRAM in algoritmi a ***scrittura-concorrente*** e ***scrittura-esclusiva***. Le abbreviazioni comunemente usate per i tipi di algoritmi che si incontreranno sono

- ***EREW***: lettura esclusiva e scrittura esclusiva (in inglese "exclusive read/exclusive write").

- ***CREW***: lettura concorrente e scrittura esclusiva ("concurrent read/exclusive write").
- ***ERCW***: lettura esclusiva e scrittura concorrente ("exclusive read/concurrent write") e
- ***CRCW***: lettura concorrente e scrittura concorrente ("concurrent read/concurrent write").

(Queste abbreviazioni sono di solito pronunciate come stringhe di lettere e non come parole.)

Tra questi tipi di algoritmi, gli estremi – EREW e CRCW – sono i più conosciuti. Una PRAM che consente l'esecuzione solo di algoritmi EREW è chiamata ***PRAM EREW*** e una che consente l'esecuzione di algoritmi CRCW è chiamata ***PRAM CRCW***. Una PRAM CRCW può, naturalmente, eseguire algoritmi EREW ma una PRAM EREW non può offrire direttamente il supporto agli accessi concorrenti alla memoria richiesti dagli algoritmi CRCW. L'hardware sottostante di una PRAM EREW è relativamente semplice, e quindi veloce, perché non necessita della gestione dei conflitti di letture e scritture concorrenti in memoria. Una PRAM CRCW richiede maggior supporto hardware se l'ipotesi sull'unità di tempo è di fornire una misura ragionevolmente accurata delle prestazioni degli algoritmi, ma fornisce un modello di programmazione che è chiaramente più semplice di quello di una PRAM EREW.

Degli altri due tipi di algoritmi – CREW e ERCW – quello CREW ha avuto maggior considerazione nelle pubblicazioni scientifiche. Da un punto di vista pratico, però, offrire il supporto alle scritture concorrenti non è più pesante che offrire il supporto alle letture concorrenti. In questo capitolo, generalmente, si tratterà un algoritmo come se fosse CRCW qualora contenga letture o scritture concorrenti, senza fare ulteriori distinzioni. Nel paragrafo 30.2, si discutono i punti più delicati di questa distinzione.

Quando più processori scrivono nella stessa locazione in un algoritmo CRCW, l'effetto della scrittura parallela non è ben definito senza una elaborazione ulteriore. In questo capitolo, si userà il modello ***CRCW comune***: quando più processori scrivono nella stessa locazione di memoria, essi devono scrivere tutti uno stesso valore. Vi sono alcuni tipi alternativi di PRAM nelle pubblicazioni scientifiche che gestiscono questo problema con ipotesi differenti. Altre scelte comprendono il modello

- ***arbitrario***: è di fatto memorizzato un valore arbitrario tra quelli scritti,
- ***a priorità***: è memorizzato il valore scritto dal processore con indice più basso
- ***combinato***: il valore memorizzato è qualche combinazione specifica dei valori scritti.

Nell'ultimo caso, la combinazione specificata è tipicamente qualche funzione associativa e commutativa come la somma (memorizza la somma di tutti i valori scritti) o il massimo (memorizza solo il massimo tra i valori scritti).

Sincronizzazione e controllo

Gli algoritmi PRAM devono essere fortemente sincronizzati per funzionare correttamente. Come si raggiunge la sincronizzazione? Ancora, i processori negli algoritmi PRAM devono spesso cercare le condizioni di terminazione dei cicli che dipendono dallo stato di tutti i processori. Come è realizzata questa funzione di controllo?

Non si discuteranno in modo estensivo questi problemi. Molti calcolatori paralleli reali impiegano una rete di controllo che connette i processori e che aiuta nella sincronizzazione e per le condizioni di terminazione. Tipicamente, la rete di controllo può realizzare queste funzioni altrettanto velocemente quanto una rete di instradamento può realizzare i riferimenti alla memoria globale.

È comunque sufficiente, ai nostri scopi, assumere che i processori siano inerentemente e strettamente sincronizzati. Tutti i processori eseguono gli stessi comandi allo stesso istante. Nessun processore si porta avanti eseguendo nuovi comandi mentre gli altri stanno ancora eseguendo vecchi comandi. Quando si descriverà il primo algoritmo parallelo, si puntualizzerà dove si assume che i processori siano sincronizzati.

Per la ricerca della terminazione di un ciclo parallelo che dipende dallo stato di tutti i processori, si assumerà che una condizione di terminazione parallela possa essere controllata attraverso la rete di controllo in tempo $O(1)$. Qualche modello PRAM EREW nelle pubblicazioni scientifiche non fa quest'ipotesi e il tempo (logaritmico) per controllare la condizione del ciclo deve essere incluso nel tempo di esecuzione totale (si veda l'Esercizio 30.1-8). Come si vedrà nel paragrafo 30.2, le PRAM CRCW non necessitano di una rete di controllo per verificare la terminazione: esse possono verificare la condizione di terminazione di un ciclo parallelo in tempo $O(1)$ attraverso l'uso di scritture concorrenti.

Struttura del capitolo

Il paragrafo 30.1 introduce la tecnica del salto dei puntatori che fornisce un modo veloce di manipolare le liste in parallelo. Si mostra come il salto dei puntatori possa essere usato per eseguire il calcolo dei prefissi su liste e come algoritmi veloci sulle liste possano essere adattati a essere usati per gli alberi. Il paragrafo 30.2 discute le potenze di calcolo relative degli algoritmi CRCW e EREW e mostra che l'accesso concorrente alla memoria fornisce maggiore potenza.

Il paragrafo 30.3 presenta il Teorema di Brent che mostra come i circuiti combinatori possano essere efficientemente simulati dalle PRAM. Il paragrafo discute anche il problema importante dell'efficienza rispetto al "lavoro" e fornisce le condizioni sotto le quali un algoritmo PRAM per p processori può essere tradotto in modo efficiente in un algoritmo PRAM per p' processori per qualunque $p' < p$. Il paragrafo 30.4 riprende il problema di eseguire un calcolo dei prefissi su una lista concatenata e mostra come un algoritmo randomizzato può eseguire il calcolo in modo efficiente. Infine, il paragrafo 30.5 mostra come si possano risolvere i conflitti nel calcolo parallelo in tempo molto meno che logaritmico usando un algoritmo deterministico.

Gli algoritmi paralleli di questo capitolo sono stati tratti principalmente dall'area della teoria dei grafi. Essi rappresentano solo una scarna selezione dell'attuale schiera di algoritmi paralleli. Le tecniche introdotte in questo capitolo, però, sono abbastanza rappresentative delle tecniche usate per algoritmi paralleli in altre aree dell'informatica.

30.1 Salto dei puntatori

Tra i più interessanti algoritmi PRAM vi sono quelli che utilizzano i puntatori. In questo paragrafo si studia una tecnica potente, chiamata salto dei puntatori, che produce algoritmi veloci per operare sulle liste. In particolare, si introduce un algoritmo in tempo $O(\lg n)$ che, dato un oggetto da una lista di n oggetti, ne calcola la distanza dalla fine della lista. Si modifica dunque questo algoritmo per eseguire un "calcolo parallelo dei prefissi" su una lista di n oggetti in tempo $O(\lg n)$. Infine, si studia una tecnica che consente di convertire molti problemi sugli alberi a problemi sulle liste, che possono così essere risolti con il salto dei

Figura 30.2 *Calcolo in una lista di n oggetti della distanza di ogni oggetto dalla fine della lista in tempo $O(\lg n)$ usando il salto dei puntatori.* (a) *Una lista concatenata rappresentata in una PRAM inizializzata con d valori. Al termine dell'algoritmo, ogni valore d contiene la distanza dell'oggetto dalla fine della lista. Il processore responsabile di ogni oggetto appare sopra l'oggetto.* (b)-(d) *I puntatori e i valori d dopo ogni iterazione del ciclo* while *nell'algoritmo* LIST-RANK.

puntatori. Tutti gli algoritmi di questo paragrafo sono algoritmi EREW: non è richiesto alcun accesso concorrente alla memoria globale.

30.1.1 Calcolo dei ranghi

Il primo algoritmo parallelo opera sulle liste. Si può memorizzare una lista in una PRAM come si memorizza in una comune RAM. Per operare in parallelo sugli oggetti della lista, però, è conveniente assegnare ad ogni oggetto un processore "responsabile". Si assumerà che vi siano tanti processori quanti sono gli oggetti della lista e che l'i-esimo processore sia responsabile dell'i-esimo oggetto. La figura 30.2(a), per esempio, mostra una lista concatenata che consiste della sequenza di oggetti $\langle 3, 4, 6, 1, 0, 5 \rangle$. Poiché vi è un processore per ogni oggetto della lista, ogni oggetto della lista può essere manipolato dal suo processore responsabile in tempo $O(1)$.

Si supponga che sia data una lista concatenata semplice L con n oggetti e che si desideri calcolare, per ogni oggetto di L, la sua distanza dalla fine della lista. Più formalmente, se *next* è il campo puntatore, si desidera calcolare un valore $d[i]$ per ogni oggetto i nella lista tale che

$$d[i] = \begin{cases} 0 & \text{se } next[i] = \text{NIL} , \\ d[next[i]] + 1 & \text{se } next[i] \neq \text{NIL} . \end{cases}$$

Il problema di calcolare i valori $d[i]$ si chiama ***problema del calcolo dei ranghi***.

Una soluzione al problema del calcolo dei ranghi è semplicemente quella di propagare all'indietro le distanze a partire dalla fine della lista. Questo metodo richiede tempo $\Theta(n)$, poiché il k-esimo oggetto dalla fine deve aspettare che i $k - 1$ oggetti che lo seguono determinino le loro distanze dalla fine prima che esso possa determinare la sua. Questa soluzione è essenzialmente un algoritmo seriale.

Una soluzione parallela efficiente, che richiede solo tempo $O(\lg n)$, è data dal seguente pseudocodice parallelo.

```
LIST-RANK(L)
1  for ogni processore i, in parallelo
2      do if next[i] = NIL
3          then d[i] ← 0
4          else d[i] ← 1
5  while esiste un oggetto i tale che next[i]≠NIL
6      do for ogni processore i, in parallelo
7          do if next[i]≠NIL
8              then d[i] ← d[i] + d[next[i]]
9                   next[i] ← next[next[i]]
```

La figura 30.2 mostra come l'algoritmo calcoli la distanza. Ogni parte della figura mostra lo stato della lista prima di un'iterazione del ciclo **while** delle linee 5-9. La parte (a) mostra la lista appena prima dell'inizializzazione. Nella prima iterazione, i primi 5 oggetti della lista hanno puntatori diversi da NIL, così che le linee 8-9 sono eseguite dai loro processori responsabili. Il risultato appare nella parte (b) della figura. Nella seconda iterazione, solo i primi 4 oggetti hanno puntatori non nulli: il risultato di questa iterazione è mostrato nella parte (c). Nella terza iterazione, si opera solo sui primi due oggetti e il risultato finale, in cui tutti gli oggetti hanno puntatore NIL, appare nella parte (d).

L'idea realizzata alla linea 9, in cui si esegue $next[i] \leftarrow next[next[i]]$ per tutti i puntatori $next[i]$ diversi da NIL, è chiamata salto dei puntatori. Si noti che i campi puntatore vengono modificati dal salto dei puntatori, distruggendo così la struttura della lista. Se la struttura della lista deve essere mantenuta, allora si effettuano le copie dei puntatori *next* e si usano le copie per calcolare le distanze.

Correttezza

LIST-RANK mantiene l'invariante che, all'inizio di ogni iterazione del ciclo **while** alle linee 5-9, per ogni oggetto i, se si sommano i valori di d nella sottolista che inizia da i, si ottiene la corretta distanza da i alla fine della lista originaria L. Nella figura 30.2(b), per esempio, la sottolista che inizia dall'oggetto 3 è la sequenza $\langle 3, 6, 0 \rangle$ i cui valori di d 2, 2 e 1 danno come somma 5, la distanza dell'oggetto 3 dalla fine della lista originaria. La ragione per cui si mantiene tale invariante è che quando un oggetto "scavalca" il suo successore nella lista, esso somma il valore di d del suo successore al proprio.

Si osservi che, perché questo algoritmo di salto dei puntatori funzioni correttamente, gli accessi paralleli alla memoria devono essere sincronizzati. Ogni esecuzione della linea 9 può aggiornare alcuni puntatori *next*. Si conta sul fatto che tutte le letture dalla memoria sul lato destro dell'assegnamento (lettura di $next[next[i]]$) si verifichino prima di qualunque scrittura nella memoria (scrittura di $next[i]$) sul lato sinistro.

Vediamo ora perché LIST-RANK è un algoritmo EREW. Poiché ogni processore è responsabile al più di un oggetto, ciascuna lettura e scrittura alle linee 2-7 è esclusiva, come lo sono

le scritture alle linee 8-9. Si osservi che il salto dei puntatori mantiene l'invariante che per qualunque coppia di oggetti distinti i e j, $next[i] \neq next[j]$ oppure $next[i] = next[j] = \text{NIL}$. Questa invariante è certamente vera per la lista iniziale ed è mantenuta dalla linea 9. Poiché tutti i valori *next* diversi da NIL sono distinti, tutte le letture alla linea 9 sono esclusive.

Non c'è la necessità di supporre che sia eseguita qualche sincronizzazione alla linea 8 se tutte le letture devono essere esclusive. In particolare, si richiede che tutti i processori i leggano $d[i]$ e poi $d[next[i]]$. Con questa sincronizzazione, se un oggetto ha $next[i] \neq \text{NIL}$ e vi è un altro oggetto j che punta ad i (cioè $next[j] = i$), allora la prima lettura carica $d[i]$ per il processore i e la seconda lettura carica $d[i]$ per il processore j. Pertanto LIST-RANK è un algoritmo EREW.

D'ora in poi, si ignoreranno questi dettagli sulla sincronizzazione e si assumerà che la PRAM e l'ambiente di programmazione si comportino in modo consistente e sincronizzato, con tutti i processori che eseguono letture e scritture nello stesso tempo.

Analisi

Mostriamo ora che se vi sono n oggetti nella lista L, allora LIST-RANK richiede tempo $O(\lg n)$. Poiché l'inizializzazione richiede tempo $O(1)$, è sufficiente mostrare che vi sono esattamente $\lceil \lg n \rceil$ iterazioni. L'osservazione chiave è che ogni passo del salto dei puntatori trasforma ogni lista in due liste che si intercalano: una consiste degli oggetti nelle posizione pari e l'altra consiste degli oggetti nelle posizioni dispari. Pertanto ogni passo del salto dei puntatori raddoppia il numero delle liste e dimezza la loro lunghezza. Dopo $\lceil \lg n \rceil$ iterazioni, allora, tutte le liste contengono solo un oggetto.

Si è fatta l'ipotesi che il controllo di terminazione alla linea 5 richieda tempo $O(1)$, presumibilmente richiesto dalla rete di controllo nella PRAM EREW. L'Esercizio 30.1-8 chiede di descrivere una realizzazione EREW di LIST-RANK che esegua, in tempo $O(\lg n)$, il controllo della terminazione esplicitamente nello pseudocodice.

Oltre al tempo di esecuzione parallelo, vi è un'altra interessante misura per gli algoritmi paralleli. Si definisce il ***lavoro*** eseguito da un algoritmo parallelo come il prodotto del suo tempo di esecuzione per il numero di processori richiesti. Intuitivamente il lavoro è la quantità di calcolo che una macchina RAM seriale esegue quando simula l'algoritmo parallelo.

La procedura LIST-RANK esegue lavoro $\Theta(n \lg n)$, poiché richiede n processori e viene eseguito in tempo $\Theta(\lg n)$. L'algoritmo seriale diretto per il problema del calcolo dei ranghi impiega tempo $O(n)$, indicando così che LIST-RANK esegue più lavoro di quello assolutamente necessario, ma solo per un fattore logaritmico.

Un algoritmo PRAM A si definisce ***efficiente rispetto al lavoro***, confrontato con un altro algoritmo B (seriale o parallelo) per lo stesso problema, se il lavoro eseguito da A è uguale al lavoro eseguito da B a meno di un fattore costante. Si dice anche più semplicemente che se un algoritmo PRAM A è ***efficiente rispetto al lavoro***, allora è efficiente rispetto al lavoro relativamente all'algoritmo migliore possibile su una RAM seriale. Poiché il miglior algoritmo seriale per il problema del calcolo dei ranghi impiega tempo $\Theta(n)$ su una RAM seriale, LIST-RANK non è efficiente rispetto al lavoro. Si presenterà un algoritmo parallelo efficiente rispetto al lavoro per il problema del calcolo dei ranghi nel paragrafo 30.4.

30.1.2 Calcolo parallelo dei prefissi su una lista

La tecnica del salto dei puntatori si estende ben oltre l'applicazione del calcolo dei ranghi. Il paragrafo 29.2.2 mostra come, nel contesto dei circuiti aritmetici, un "calcolo dei prefissi" possa essere usato per eseguire velocemente somme binarie. Si studierà come il salto dei puntatori possa essere usato per eseguire il calcolo dei prefissi. L'algoritmo EREW per il calcolo dei prefissi impiega tempo $O(\lg n)$ su liste di n oggetti.

Un ***calcolo dei prefissi*** è definito in termini dell'operatore binario associativo $\otimes$. Il calcolo richiede come input una sequenza $\langle x_1, x_2, \ldots, x_n \rangle$ e produce come output una sequenza $\langle y_1, y_2, \ldots, y_n \rangle$ tale che $y_1 = x_1$ e

$$\begin{aligned} y_k &= y_{k-1} \otimes x_k \\ &= x_1 \otimes x_2 \otimes \ldots \otimes x_k \end{aligned}$$

per $k = 2, 3, \ldots, n$. In altre parole, ogni y_k è ottenuto dalla "moltiplicazione" tra i primi k elementi della sequenza, da cui il termine "prefisso". (La definizione nel Capitolo 29 indicizza la sequenza a partire da 0, mentre questa definizione indicizza da 1 – una differenza non essenziale.)

Come esempio di calcolo dei prefissi, si supponga che ogni elemento di una lista di n oggetti contenga il valore 1 e sia $\otimes$ la normale somma. Poiché il k-esimo elemento della lista contiene il valore $x_k = 1$ per $k = 1, 2, \ldots, n$, un calcolo dei prefissi produce $y_k = k$, l'indice del k-esimo elemento. Pertanto, un altro modo di risolvere il problema del calcolo dei ranghi è di invertire la lista (che può essere fatto in tempo $O(1)$), eseguire il calcolo dei prefissi e sottrarre 1 da ogni valore calcolato.

Si mostra ora come un algoritmo EREW possa calcolare i prefissi in parallelo in tempo $O(\lg n)$ su liste di n oggetti. Per comodità, si definisce la notazione

$$[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$$

per interi i e j tali che $1 \le i \le j \le n$. Allora

$[k, k] = x$ per $k = 1, 2, \ldots, n$, e

$[i, k] = [i, j] \otimes [j + 1, k]$ per $1 \le i \le j < k \le n$.

Utilizzando questa notazione, l'obiettivo di un calcolo dei prefissi è di calcolare $y_k = [1, k]$ per $k = 1, 2, \ldots, n$.

Quando si esegue un calcolo dei prefissi su una lista, si desidera che l'ordine della sequenza di input $\langle x_1, x_2, \ldots, x_n \rangle$ sia determinato dal modo in cui gli oggetti sono collegati insieme nella lista e non dall'indice dell'oggetto nell'array di memoria che memorizza gli oggetti. (L'Esercizio 30.1-2 richiede un algoritmo dei prefissi per gli array.) Il seguente algoritmo EREW comincia con un valore $x[i]$ in ogni oggetto i in una lista L. Se l'oggetto i è il k-esimo oggetto dall'inizio della lista, allora $x[i] = x_k$ è il k-esimo elemento della sequenza di input. Pertanto, il calcolo parallelo dei prefissi produce $y[i] = y_k = [1, k]$.

```
LIST-PREFIX(L)
1  for ogni processore i, in parallelo
2          do y[i] ← x[i]
3  while esiste un oggetto i tale che next[i] ≠ NIL
4          do for ogni processore i, in parallelo
```

Figura 30.3 *L'algoritmo parallelo dei prefissi* LIST-PREFIX *su una lista concatenata.* (a) *Il valore iniziale y del k-esimo oggetto nella lista oppure è* $[k, k]$. *Il puntatore next del k-esimo oggetto punta al* $(k + 1)$*-esimo oggetto oppure è* NIL *per l'ultimo oggetto.* (b)-(d) *I valori y e next prima di ogni controllo della linea 3. La risposta finale è nella parte (d), in cui il valore y per il k-esimo oggetto è* $[1, k]$ *per tutti i k.*

```
5          do if next[i] ≠ NIL
6                  then  y[next[i]] ← y[i] ⊗ y[next[i]]
7                        next[i] ← next[next[i]]
```

Lo pseudocodice e la figura 30.3 indicano la somiglianza tra questo algoritmo e LIST-RANK. Le uniche differenze sono l'inizializzazione e l'aggiornamento dei valori d o y. Nella LIST-RANK, il processore i aggiorna $d[i]$ – il suo valore d – laddove in LIST-PREFIX, il processore i aggiorna $y[next[i]]$ – il valore y di un altro processore. Si noti che LIST-PREFIX è EREW per le stesse ragioni di LIST-RANK: il salto dei puntatori mantiene l'invariante che per oggetti distinti i e j, $next[i] \ne next[j]$ oppure $next[i] = next[j] = \text{NIL}$.

La figura 30.3 mostra lo stato della lista prima di ogni iterazione del ciclo while. La procedura mantiene l'invariante che alla fine della t-esima esecuzione del ciclo while il k-esimo processore memorizza $[\max(1, k - 2^t + 1), k]$, per $k = 1, 2, \ldots, n$. Nella prima iterazione, il k-esimo oggetto della lista punta inizialmente al $(k + 1)$-esimo oggetto, tranne l'ultimo che ha un puntatore NIL. La linea 6 fa sì che il k-esimo oggetto, per $k = 1, 2, \ldots, n - 1$, carichi il valore $[k + 1, k + 1]$ dal suo successore. Quindi esegue l'operazione $[k, k] \otimes [k + 1, k + 1]$, producendo $[k, k + 1]$ che viene memorizzato nel suo successore. I puntatori *next* sono saltati come in LIST-RANK e il risultato della prima iterazione appare nella figura 30.3(b). Si può vedere similmente la seconda iterazione. Per $k = 1, 2, \ldots, n - 2$, il k-esimo oggetto carica il valore $[k + 1, k + 2]$ dal suo successore (come definito dal nuovo valore nel suo campo *next*) e quindi memorizza $[k - 1, k] \otimes [k + 1, k + 2] = [k + 1, k + 2]$ nel suo successore. Il risultato è mostrato nella figura 30.3(c). Nella terza e ultima iterazione, solo i primi due oggetti della lista hanno il puntatore diverso da NIL e caricano i valori dai loro successori nelle loro rispettive liste. Il risultato finale appare nella figura 30.3(d). La proprietà fondamentale che fa funzionare LIST-PREFIX è che ad ogni passo, se si esegue un calcolo dei prefissi su ogni lista esistente, ogni oggetto ottiene il suo valore corretto.

Poiché i due algoritmi usano lo stesso meccanismo del salto dei puntatori, LIST-PREFIX ha la stessa analisi di LIST-RANK: il tempo di esecuzione è $O(\lg n)$ su una PRAM EREW e il lavoro totale eseguito è $\Theta(n \lg n)$.

30.1.3 La tecnica del ciclo euleriano

In questo paragrafo si introdurrà la tecnica del ciclo euleriano e si mostrerà come possa essere applicata al problema di calcolare la profondità di ogni nodo in un albero binario di n nodi. Uno dei passi chiave di questo algoritmo EREW con tempo $O(\lg n)$ è il calcolo parallelo dei prefissi.

Per memorizzare alberi binari in una PRAM, si usa una semplice rappresentazione ad albero binario dell'ordinamento presentata nel paragrafo 11.4. Ogni nodo i ha campi *parent*[i], *left*[i] e *right*[i] che puntano rispettivamente al nodo padre, al fratello sinistro e al fratello destro di i. Si assuma che ogni nodo sia identificato da un intero non negativo. Per ragioni che diventeranno subito chiare, si associano tre processori ad ogni nodo piuttosto che uno solo; questi processori saranno chiamati i processori A, B e C del nodo. Dovrebbe essere semplice individuare la corrispondenza tra un nodo e i suoi tre processori; per esempio, il nodo i potrebbe essere associato con i processori $3i$, $3i + 1$ e $3i + 2$.

Il calcolo della profondità di ogni nodo in un albero di n nodi richiede tempo $O(n)$ su una RAM seriale. Un algoritmo parallelo semplice per calcolare le profondità propaga un "onda" dalla radice dell'albero verso il basso. L'onda raggiunge simultaneamente tutti i nodi alla stessa profondità: così, incrementando un contatore che va avanti con l'onda, si può calcolare la profondità di ogni nodo. Questo algoritmo parallelo funziona bene su un albero binario completo poiché impiega un tempo proporzionale all'altezza dell'albero. L'altezza dell'albero, però, potrebbe essere pari a $n - 1$, nel qual caso l'algoritmo dovrebbe impiegare tempo $\Theta(n)$ – non migliore dell'algoritmo seriale. Usando la tecnica del ciclo euleriano, però, si possono calcolare su una PRAM EREW le profondità dei nodi in tempo $O(\lg n)$, qualunque sia l'altezza dell'albero.

Un *ciclo euleriano* di un grafo è un ciclo che attraversa ogni arco esattamente una volta anche se può visitare lo stesso vertice più di una volta. Dal Problema 23-3, un grafo orientato e connesso ha un ciclo euleriano se e solo se, per tutti i vertici v, il grado d'entrata di v è uguale al grado d'uscita di v. Poiché ogni arco non orientato (u, v) in un grafo non orientato corrisponde a due archi orientati (u, v) e (v, u) nella versione orientata, la versione orientata di qualunque grafo non orientato e connesso – e dunque di qualunque albero non orientato – ha un ciclo euleriano.

Per calcolare le profondità dei nodi in un albero binario T, si forma prima un ciclo euleriano della versione orientata di T (visto come un grafo non orientato). Il cammino corrisponde a una visita dell'albero ed è rappresentato nella figura 30.4(a) da una lista concatenata che attraversa i nodi dell'albero. La sua struttura è la seguente:

- Il processore A di un nodo punta al processore A del figlio sinistro, se esiste, altrimenti punta al proprio processore B.
- Il processore B di un nodo punta al processore A del figlio destro, se esiste, altrimenti punta al proprio processore C.
- Il processore C di un nodo punta al processore B del padre se è un figlio sinistro e al processore C del padre se è un figlio destro. Il processore C della radice punta a NIL.

Pertanto, l'inizio della lista concatenata formata da un ciclo euleriano è il processore A della radice e la fine è il processore C della radice. Dati i puntatori che compongono l'albero originario, un ciclo euleriano può essere costruito in tempo $O(1)$.

Figura 30.4 *Uso della tecnica del ciclo euleriano per calcolare la profondità di ogni nodo in un albero binario.* (a) *Il ciclo euleriano è una lista che corrisponde ad una visita dell'albero. Ogni processore contiene un numero usato nel calcolo parallelo dei prefissi per calcolare la profondità dei nodi.* (b) *Il risultato del calcolo parallelo dei prefissi sulla lista concatenata della parte* (a). *Il processore C di ogni nodo (in nero) contiene la profondità del nodo. (Si può verificare il risultato di questo calcolo dei prefissi calcolandolo in modo seriale.)*

Quando si ha la lista concatenata che rappresenta il ciclo euleriano di T, si mette un 1 in ogni processore A, uno 0 in ogni processore B e un -1 in ogni processore C, come mostrato nella figura 30.4(a). Quindi si esegue un calcolo parallelo dei prefissi usando come operazione associativa la normale somma, come si è già fatto nel paragrafo 30.1.2. La figura 30.4(b) mostra il risultato del calcolo parallelo dei prefissi.

Si è affermato che dopo l'esecuzione del calcolo parallelo dei prefissi, la profondità di ogni nodo risiede nel processore C del nodo. Perché? I numeri sono messi nei processori A, B e C in modo tale che l'effetto di visitare un sottoalbero è di sommare 0 alla somma corrente. Il processore A di ogni nodo i contribuisce con 1 alla somma corrente nel sottoalbero sinistro di i, ciò riflette il fatto che la profondità del figlio sinistro di i sia più grande di 1 della profondità di i. Il processore B contribuisce con 0 perché la profondità del figlio sinistro del nodo i è uguale alla profondità del figlio destro del nodo i. Il processore C contribuisce con -1, in modo che dalla prospettiva del padre del nodo i, l'intera visita del sottoalbero radicato nel nodo i non abbia effetti sulla somma corrente.

La lista che rappresenta il ciclo euleriano può essere calcolata in tempo $O(1)$. Essa ha $3n$ oggetti e così il calcolo parallelo dei prefissi richiede soltanto tempo $O(\lg n)$. Pertanto, la quantità totale di tempo per calcolare le profondità di tutti i nodi è $O(\lg n)$. Poiché non sono necessari accessi concorrenti alla memoria, l'algoritmo è un algoritmo EREW.

Esercizi

30.1-1 Si dia un algoritmo EREW che, per ogni oggetto in una lista di n oggetti, determini in tempo $O(\lg n)$ se è l'oggetto centrale (cioè l'$\lfloor n/2 \rfloor$-esimo).

30.1-2 Si dia un algoritmo EREW che esegua in tempo $O(\lg n)$ il calcolo dei prefissi su un array $x[1..n]$. Non usare puntatori, ma eseguire direttamente il calcolo degli indici.

30.1-3 Si supponga che ogni oggetto in una lista L di n oggetti sia colorato di rosso o di blu. Si dia un algoritmo EREW efficiente per formare due liste con gli oggetti di L: una contenente gli oggetti blu e una contenente gli oggetti rossi.

30.1-4 Una PRAM EREW ha n oggetti distribuiti su alcune liste circolari disgiunte. Si dia un algoritmo efficiente che individui per ogni lista un oggetto arbitrario che la rappresenta e informi ogni oggetto della lista dell'identità del rappresentante. Si assuma che ogni processore conosca il proprio indice e che tale indice sia unico.

30.1-5 Si dia un algoritmo EREW che calcoli in tempo $O(\lg n)$ la dimensione del sottoalbero radicato in ogni nodo di un albero binario di n nodi. (*Suggerimento:* si mantenga la differenza di due valori in una somma corrente lungo un ciclo euleriano.)

30.1-6 Si dia un algoritmo EREW che calcoli la sequenza di nodi ottenuta visitando un albero binario arbitrario in ordine anticipato, simmetrico e differito.

30.1-7 Si estenda la tecnica del ciclo euleriano dagli alberi binari agli alberi ordinati con grado arbitrario dei nodi. In particolare, si descriva una rappresentazione per alberi ordinati che consenta di applicare la tecnica del ciclo euleriano. Si dia un algoritmo EREW che calcoli in tempo $O(\lg n)$ la profondità dei nodi di un albero ordinato di n nodi.

30.1-8 Si descriva una realizzazione EREW di LIST-RANK che esegua esplicitamente in tempo $O(\lg n)$ il controllo della terminazione dei cicli. (*Suggerimento:* intercalare il controllo con il corpo del ciclo.)

30.2 Confronto tra algoritmi EREW e algoritmi CRCW

La discussione sul fatto che gli accessi concorrenti alla memoria debbano essere forniti dall'hardware di un calcolatore parallelo è controversa. Alcuni dicono che i meccanismi hardware di supporto agli algoritmi CRCW sono troppo costosi e usati troppo poco frequentemente per essere giustificati. Altri lamentano il fatto che le PRAM EREW forniscono un modello di programmazione troppo restrittivo. Le risposte a questa discussione probabilmente stanno a metà, per cui sono stati proposti vari modelli di compromesso. Nonostante ciò è istruttivo esaminare quale vantaggio algoritmico è fornito dagli accessi concorrenti alla memoria.

In questo paragrafo, si mostrerà che vi sono problemi per cui un algoritmo CRCW supera nelle prestazioni il migliore algoritmo EREW. Per il problema di trovare l'identità delle radici degli alberi di una foresta, le letture concorrenti consentono un algoritmo più veloce. Per il problema di trovare il massimo elemento in un array, le scritture concorrenti permettono un algoritmo più veloce.

Un problema in cui le letture concorrenti aiutano

Si supponga che sia data una foresta di alberi binari in cui ogni nodo i ha un puntatore $parent[i]$ a suo padre e si desideri che ogni nodo conosca l'identità della radice del suo albero. Associando il processore i ad ogni nodo i nella foresta F, il seguente algoritmo con salto dei puntatori memorizza l'identità della radice dell'albero di ogni nodo i in $root[i]$.

```
FIND-ROOTS(F)
for ogni processore i, in parallelo
    do if parent[i] = NIL
        then root[i] ← i
while esiste un nodo i tale che parent[i] ≠ NIL
    do for ogni processore i, in parallelo
        do if parent[i] ≠ NIL
            then  if parent[parent[i]] = NIL then root[i] ← root[parent[i]]
                  parent[i] ← parent[parent[i]]
```

Figura 30.5 *Identificazione delle radici in una foresta di alberi binari con una CREW PRAM. I numeri dei nodi sono vicini ai nodi e i campi root memorizzati appaiono dentro i nodi. Le frecce rappresentano i puntatori parent.* (a)-(d) *Lo stato degli alberi nella foresta ogni volta che viene eseguita la linea* 4 *di* FIND-ROOTS. *Si noti che le lunghezze del cammino sono dimezzate ad ogni iterazione.*

La figura 30.5 illustra il comportamento di questo algoritmo. Dopo l'inizializzazione eseguita nelle linee 1-3, mostrata nella figura 30.5(a), gli unici nodi che conoscono la loro radice sono le radici stesse. Il ciclo **while** delle linee 4-8 esegue il salto dei puntatori e riempie i campi *root*. Le figure 30.5(b)-(d) mostrano lo stato della foresta dopo la prima, la seconda e la terza iterazione del ciclo. Come si può vedere, l'algoritmo mantiene l'invariante che se *parent*[*i*] = NIL, allora a *root*[*i*] è stata assegnata l'identità della radice del nodo.

Si può affermare che FIND-ROOTS è un algoritmo CREW che impiega tempo $O(\lg d)$, dove d è la profondità dell'albero con profondità massima della foresta. Le uniche scritture si verificano nelle linee 3, 7 e 8 e sono tutte esclusive poiché, per ognuna, il processore i scrive

solo nel nodo i. Però le letture nelle linee 7-8 sono concorrenti perché alcuni nodi possono avere puntatori allo stesso nodo. Nella figura 30.5(b), per esempio, si vede che durante la seconda iterazione del ciclo **while**, *root*[4] e *parent*[4] sono letti dai processori 18, 2 e 7.

Il tempo di esecuzione di FIND-ROOTS è $O(\lg d)$ essenzialmente per le stesse ragioni che valgono per LIST-RANK: la lunghezza di ogni cammino è dimezzata ad ogni iterazione. La figura 30.5 mostra chiaramente questa caratteristica.

Quanto velocemente n nodi in una foresta possono determinare le radici dei loro alberi binari usando solo letture esclusive? Con una semplice argomentazione si mostra che è richiesto tempo $\Omega(\lg n)$. L'osservazione chiave è che quando le letture sono esclusive, ogni passo della PRAM permette che un dato pezzo di informazione sia copiato al massimo in un'altra locazione di memoria; in tal modo il numero di locazioni che possono contenere un dato pezzo di informazione al più raddoppia ad ogni passo. Osservando un singolo albero, inizialmente si ha che al più una locazione di memoria memorizza l'identità della radice. Dopo un passo, al più 2 locazioni di memoria possono contenere l'identità della radice; dopo k passi, al più 2^{k-1} locazioni possono contenere l'identità della radice. Se la dimensione dell'albero è $\Theta(n)$, saranno necessarie $\Theta(n)$ locazioni per contenere l'identità della radice al termine dell'algoritmo; pertanto, sono richiesti in tutto $\Omega(\lg n)$ passi.

Se la profondità d dell'albero di massima profondità nella foresta è $2^{n/\lg n}$, l'algoritmo CREW FIND-ROOTS supera asintoticamente qualunque algoritmo EREW. In particolare, per qualunque foresta di n nodi il cui albero con profondità massima sia un albero binario bilanciato con $\Theta(n)$ nodi, $d = O(\lg n)$, nel qual caso FIND-ROOTS impiega tempo $O(\lg \lg n)$. Qualunque algoritmo EREW per questo problema deve impiegare tempo $\Omega(\lg n)$ che è asintoticamente più lento. Pertanto, le letture concorrenti aiutano nella risoluzione di questo problema. L'Esercizio 30.2-1 dà uno scenario più semplice in cui le letture concorrenti aiutano.

Un problema in cui le scritture concorrenti aiutano

Per dimostrare che scritture concorrenti offrono un vantaggio nelle prestazioni rispetto alle scritture esclusive, si esamina il problema di trovare il massimo elemento in un array di numeri reali. Si vedrà che qualunque algoritmo EREW per questo problema richiede tempo $\Omega(\lg n)$ e che nessun algoritmo CREW si comporta meglio. Il problema può essere risolto in tempo $O(1)$ usando un comune algoritmo CRCW in cui quando più processori scrivono nella stessa locazione, scrivono tutti lo stesso valore.

L'algoritmo CRCW che trova il massimo in un array di n elementi assume che l'array di input sia $A[0..n-1]$. L'algoritmo usa n^2 processori, dove ogni processore confronta $A[i]$ con $A[j]$ per qualche i e j con $0 \le i, j \le n-1$. In effetti, l'algoritmo esegue una matrice di confronti: è possibile identificare ognuno degli n^2 processori coinvolti mediante una coppia di indici (i, j).

```
FAST-MAX(A)
n ← length[A]
for i ← 0 to n − 1, in parallelo
        do m[i] ← TRUE
for i ← 0 to n − 1 e j ← 0 to n − 1, in parallelo
```

		$A[j]$					
		5	6	9	2	9	m
	5	F	T	T	F	T	F
	6	F	F	T	F	T	F
$A[i]$	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 *Ricerca del massimo di n valori in tempo $O(1)$ utilizzando l'algoritmo* CRCW FAST-MAX. *Per ogni coppia ordinata di elementi nell'array di input $a = \langle 5, 6, 9, 2, 9 \rangle$, il risultato del confronto $A[i] < A[j]$ è mostrato nella matrice dove* T *sta per* TRUE ed F *sta per* FALSE. *Per qualsiasi riga contenente un valore* TRUE *il corrispondente elemento di m, mostrato sulla destra è posto a* FALSE. *Gli elementi di m che contengono* TRUE *corrispondono agli elementi di A più grandi. In questo caso, il valore 9 è assegnato alla variabile max.*

```
5        do if A[i] < A[j]
6                then m[i] ← FALSE
7  for i ← 0 to n − 1, in parallelo
8        do if m[i] = TRUE
9                then max ← A[i]
10 return max
```

La linea 1 semplicemente determina la lunghezza dell'array A; è necessario soltanto che venga eseguita da un solo processore, per esempio il processore 0. Si usa un array $m[0..n-1]$, dove il processore i è responsabile di $m[i]$. Si vuole $m[i]$ = TRUE se e solo se $A[i]$ è il massimo valore nell'array A. Si comincia (linee 2-3) con l'assumere che ogni elemento dell'array possa essere il massimo e ci si basa sul confronto alla linea 5 per determinare quali elementi dell'array non sono il massimo.

La figura 30.6 illustra la parte rimanente dell'algoritmo. Nel ciclo alle linee 4-6, si controlla ogni coppia ordinata di elementi dell'array A. Per ogni coppia $A[i]$ e $A[j]$, la linea 5 controlla se $A[i] < A[j]$. Se la risposta è TRUE, si sa che $A[i]$ non può essere il massimo e la linea 6 assegna $m[i] \leftarrow$ FALSE per registrare questo fatto. Alcune coppie (i, j) possono scrivere simultaneamente in $m[i]$ ma tutti scriveranno lo stesso valore: FALSE.

Dopo che la linea 6 è stata eseguita, $m[i]$ = TRUE esattamente per gli indici i tali che $A[i]$ è il massimo. Le linee 7-9 mettono il valore massimo nella variabile max, restituita alla linea 10. Più processori possono scrivere nella variabile max, ma se lo fanno, scrivono tutti lo stesso valore, consistentemente con il modello PRAM CRCW comune.

Poiché tutti e tre i cicli dell'algoritmo sono eseguiti in parallelo, FAST-MAX impiega tempo $O(1)$. Naturalmente non è efficiente rispetto al lavoro, poiché richiede n^2 processori e il problema di trovare il massimo numero in un array può essere risolto da un algoritmo seriale con tempo $\Theta(n)$. Tuttavia, è possibile avvicinarsi ad un algoritmo efficiente rispetto al lavoro come richiede di mostrare l'Esercizio 30.2-6.

In un certo senso, la chiave di FAST-MAX è che una PRAM CRCW è capace di eseguire un AND booleano di n variabili in tempo $O(1)$ con n processori. (Poiché questa possibilità è prevista nel modello CRCW comune, si mantiene anche nel modello PRAM CRCW che è più potente.) Il codice in effetti esegue più AND in una sola volta, calcolando per $i = 0, 1, \ldots, n-1$,

$$m[i] = \bigwedge_{j=0}^{n-1} (A[i] \geq A[j]) ,$$

che può essere derivata dalle leggi di DeMorgan (5.2). Questa potente possibilità dell'AND può essere usata in altri modi. Per esempio, l'opportunità per una PRAM CRCW di eseguire un AND in tempo $O(1)$ elimina la necessità di una rete di controllo separata per verificare se tutti i processori hanno finito di iterare un ciclo, come si è ipotizzato per gli algoritmi EREW. La decisione di terminare un ciclo è semplicemente l'AND delle richieste di tutti i processori di terminare il ciclo.

Il modello EREW non ha questa potente possibilità dell'AND. Qualunque algoritmo EREW che calcola il massimo di n elementi richiede tempo $\Omega(\lg n)$. La dimostrazione concettualmente è analoga agli argomenti sul limite inferiore usati per la ricerca della radice di un albero binario. In quella prova, si verificò quanti nodi potessero "conoscere" l'identità della radice e si mostrò che al massimo il numero di nodi raddoppiava ad ogni passo. Per il problema del calcolo del massimo di n elementi, si consideri il numero di elementi che "pensano" di potere essere il massimo. Intuitivamente, ad ogni passo di una PRAM EREW, questo numero al più può dimezzarsi, il che porta al limite inferiore $\Omega(\lg n)$.

È importante osservare che il limite inferiore $\Omega(\lg n)$ per calcolare il massimo vale anche se si consentono letture concorrenti, cioè si mantiene per algoritmi CREW. Cook, Dwork e Reischuck [50] mostrano, in effetti, che qualunque algoritmo CREW per trovare il massimo di n elementi deve impiegare tempo $\Omega(\lg n)$, anche con un numero illimitato di processori e con una memoria illimitata. Il limite inferiore si mantiene anche per il problema di calcolare l'AND di n valori booleani.

Simulazione di un algoritmo CRCW con un algoritmo EREW

Si sa ora che un algoritmo CRCW può risolvere alcuni problemi più rapidamente di un algoritmo EREW, inoltre qualunque algoritmo EREW può essere eseguito su una PRAM CRCW. Pertanto il modello CRCW è strettamente più potente del modello EREW. Ma di quanto è più potente? Nel paragrafo 30.3 si mostrerà che una PRAM EREW con p processori può ordinare p numeri in tempo $O(\lg p)$. Si usa ora questo risultato per fornire un limite superiore teorico sulla potenza di una PRAM CRCW rispetto a una PRAM EREW.

Teorema 30.1

Un algoritmo CRCW per p processori può essere al più $O(\lg p)$ volte più veloce del miglior algoritmo EREW per p processori che risolve lo stesso problema.

Dimostrazione. La dimostrazione consiste in una prova di simulazione. Si simula ogni passo dell'algoritmo CRCW con un calcolo EREW di tempo $O(\lg p)$. Poiché la potenza di calcolo di entrambe le macchine è la stessa, bisogna concentrarsi solo sugli accessi alla memoria. In questa sede si presenta la dimostrazione per la simulazione soltanto delle scritture concorrenti. La realizzazione per la simulazione delle letture è lasciata come esercizio (Esercizio 30.2-8).

I p processori nella PRAM EREW simulano una scrittura concorrente dell'algoritmo CRCW usando un array ausiliario A di lunghezza p. La figura 30.7 illustra l'idea. Quando

Figura 30.7 *Simulazione di una scrittura concorrente su una PRAM EREW.* (a) *Un passo di un algoritmo CREW comune in cui 6 processori scrivono concorrentemente nella memoria globale.* (b) *Simulazione del passo su una PRAM EREW. Le coppie ordinate contenenti locazioni e dati sono scritte in un array A. L'array è quindi ordinato. Attraverso il confronto di elementi adiacenti nell'array ci si assicura che sia realizzata nella memoria globale solo la prima scrittura di ogni gruppo di scritture identiche. In questo caso, i processori P_0, P_2, e P_5 eseguono la scrittura.*

un processore CRCW P_i, per $i = 0, 1, \ldots, p-1$, richiede di scrivere un dato x_i nella locazione l_i, ogni processore EREW P_i corrispondente scrive invece la coppia ordinata (l_i, x_i) nella locazione $A[i]$. Queste scritture sono esclusive poiché ogni processore scrive su una diversa locazione di memoria. Quindi l'array A viene ordinato rispetto al primo elemento della coppia in tempo $O(\lg p)$, il che consente che tutti i dati da mettere nella stessa locazione siano raggruppati consecutivamente.

Ogni processore EREW P_i, per $i = 0, 1, \ldots, p-1$, controlla se $A[i] = (l_j, x_j)$ e $A[i-1] = (l_k, x_k)$, dove j e k sono valori tali che $0 \le j, k \le p-1$. Se $l_j \ne l_k$ o $i = 0$, allora il processore P_i, per $i = 0, 1, \ldots, p-1$, scrive il dato x_j nella locazione l_i della memoria globale. Altrimenti il processore non fa niente. Poiché l'array A è ordinato rispetto al primo elemento della coppia, di fatto in una data locazione si effettua solo una delle scritture dei processori, così la scrittura è esclusiva. Questo processo realizza così ogni passo delle scritture concorrenti del modello CRCW comune in tempo $O(\lg p)$. ■

Altri modelli per scritture concorrenti possono essere simulati allo stesso modo. (Si veda l'Esercizio 30.2-9.)

Sorge dunque la questione di quale modello sia preferibile – CRCW o EREW – e se CRCW, quale modello CRCW. I sostenitori dei modelli CRCW fanno notare che questi sono più facili da programmare del modello EREW e che i loro algoritmi vengono eseguiti più velocemente. I critici controbattono che l'hardware per realizzare operazioni concorrenti sulla memoria è più lento dell'hardware che realizza operazioni esclusive sulla memoria, pertanto il tempo di esecuzione più veloce degli algoritmi CRCW è fittizio. In realtà, essi dicono, non si può trovare il massimo di n valori in tempo $O(1)$.

Altri dicono che la PRAM – sia EREW che CRCW – è un modello completamente sbagliato. I processori devono essere interconnessi da una rete di comunicazione e la rete di comunicazione dovrebbe fare parte del modello. I processori dovrebbero essere solo capaci di comunicare con i loro vicini nella rete.

È abbastanza chiaro che la questione del "giusto" modello parallelo non si sta orientando facilmente su alcun modello. La cosa importante da capire, però, è che questi modelli non sono altro che modelli. In una data situazione del mondo reale, i vari modelli si applicano con gradi diversi. Il grado con cui il modello corrisponde alla situazione ingegneristica è il grado con cui l'analisi algoritmica nel modello prevederà i fenomeni del mondo reale. Quindi è importante studiare i vari modelli e algoritmi paralleli, così che il campo del calcolo parallelo progredisca e possa emergere un accordo di più largo respiro sul quale paradigmi del calcolo parallelo siano più adatti ad essere realizzati.

Esercizi

30.2-1 Si supponga di sapere che una foresta di alberi binari consista di un solo albero con n nodi. Si mostri che con questa ipotesi, una realizzazione CREW di FIND-ROOTS può impiegare tempo $O(1)$, indipendentemente dalla profondità dell'albero. Si spieghi perché qualunque algoritmo EREW richiede tempo $\Omega(\lg n)$.

30.2-2 Si dia un algoritmo EREW per FIND-ROOTS che impieghi tempo $O(\lg n)$ su una foresta di n nodi.

30.2-3 Si dia un algoritmo CRCW su n processori che possa calcolare l'OR di n valori booleani in tempo $O(1)$.

30.2-4 Si descriva un efficiente algoritmo CRCW per moltiplicare due matrici booleane $n \times n$ usando n^3 processori.

30.2-5 Si descriva un algoritmo EREW per moltiplicare in tempo $O(\lg n)$ due matrici $n \times n$ di numeri reali usando n^3 processori. Vi è un algoritmo CRCW comune più veloce? Vi è un algoritmo più veloce per uno dei modelli CRCW più forti?

* *30.2-6* Si provi che per qualunque costante $\varepsilon > 0$, vi è un algoritmo CRCW di tempo $O(1)$ che usando $O(n^{1+\varepsilon})$ processori trova il massimo elemento di un array di n elementi.

* *30.2-7* Si mostri come fondere due array ordinati, ognuno di n numeri, in tempo $O(1)$ usando un algoritmo CRCW a priorità. Si descriva come usare questo algoritmo per ordinare in tempo $O(\lg n)$. L'algoritmo di ordinamento è efficiente rispetto al lavoro?

30.2-8 Si completi la prova del Teorema 30.1 descrivendo come una lettura concorrente su una PRAM CRCW di p processori sia realizzata in tempo $O(\lg p)$ su una PRAM EREW di p processori.

30.2-9 Si mostri come una PRAM EREW di p processori possa realizzare una PRAM CRCW combinata con una perdita nelle prestazioni di solo $O(\lg p)$. (*Suggerimento:* si usi il calcolo parallelo dei prefissi.)

30.3 Il Teorema di Brent e l'efficienza rispetto al lavoro

Il Teorema di Brent mostra come si possa simulare efficientemente un circuito combinatorio con una PRAM. Usando questo teorema, si possono adattare al modello PRAM molti dei risultati ottenuti per le reti di ordinamento del Capitolo 28 e per i circuiti aritmetici del Capitolo 29. I lettori che non hanno confidenza con i circuiti combinatori potrebbero rivedere il paragrafo 29.1.

Un ***circuito combinatorio*** è una rete aciclica di ***elementi combinatori***. Ogni elemento combinatorio ha uno o più input e, in questo paragrafo, si assumerà che ogni elemento abbia esattamente un output. (Elementi combinatori con $k > 1$ output possono essere considerati come k elementi separati.) Il numero di input è il ***numero di collegamenti di ingresso*** dell'elemento e il numero di posti in cui i suoi output caricano è il suo ***numero di collegamenti di uscita***. Generalmente si assumerà in questo paragrafo che il numero di collegamenti in ingresso di ogni elemento combinatorio nel circuito sia limitato da $O(1)$. Si può però avere un numero illimitato di collegamenti in ingresso.

La ***dimensione*** di un circuito combinatorio è il numero di elementi combinatori che contiene. Il numero di elementi combinatori sul cammino più lungo da un input del circuito a un output di un elemento combinatorio è la ***profondità*** dell'elemento. La profondità dell'intero circuito è la profondità massima dei suoi elementi.

Teorema 30.2 (Teorema di Brent)

Qualunque circuito combinatorio di profondità d e dimensione n con un numero di collegamenti in ingresso limitato può essere simulato da un algoritmo CREW su p processori in tempo $O(n/p + d)$.

Dimostrazione. Si memorizzano gli input del circuito combinatorio nella memoria globale della PRAM e, per ogni elemento combinatorio del circuito, si riserva una locazione per memorizzare il suo valore di output quando sarà calcolato. Un dato elemento combinatorio può

Figura 30.8 *Il Teorema di Brent. Il circuito combinatorio di dimensione 15 e profondità 5 è simulato da una PRAM CREW con 2 processori in $9 \le 15/2 + 5$ passi. La simulazione procede dall'alto in basso attraverso il circuito. I gruppi di elementi del circuito evidenziati in grigio indicano quali elementi devono essere simulati nello stesso tempo e ogni gruppo è etichettato con un numero corrispondente al tempo in cui i suoi elementi saranno simulati.*

quindi essere simulato da un solo processore PRAM in tempo $O(1)$ nel modo seguente. Il processore, semplicemente, legge i valori di input per gli elementi dai valori in memoria corrispondenti agli input dei circuiti o agli elementi di output che li forniscono, simulando così i fili del circuito. Quindi calcola la funzione degli elementi combinatori e scrive il risultato nella posizione appropriata in memoria. Poiché il numero di collegamenti in ingresso di ogni elemento del circuito è limitato, ogni funzione può essere calcolata in tempo $O(1)$.

Quindi bisogna trovare un assegnamento dei p processori della PRAM tale che tutti gli elementi combinatori siano simulati in tempo $O(n/p + d)$. Il vincolo principale è che un elemento non può essere simulato finché non siano stati calcolati gli output di qualunque elemento che carica l'input. Letture concorrenti sono impiegate quando più elementi combinatori che devono essere simulati in parallelo richiedono lo stesso valore.

Poiché tutti gli elementi a profondità 1 dipendono solo dagli input del circuito, essi sono i soli che possano essere simulati inizialmente. Dopo che essi sono stati simulati, possono essere simulati tutti gli elementi a profondità 2 e così via, finché si finisce con tutti gli elementi a profondità d. L'idea chiave è che se tutti gli elementi da profondità 1 a profondità i sono stati

simulati, si può simulare in parallelo qualunque sottoinsieme degli elementi di profondità $i+1$, poiché i loro calcoli sono indipendenti l'uno dall'altro.

Dunque la strategia di assegnamento dei processori è abbastanza semplice. Si simulano tutti gli elementi di profondità i prima di procedere a simulare quelli di profondità $i+1$. Per una data profondità i, si simulano p elementi alla volta. La figura 30.8 illustra tale strategia per $p = 2$.

Si analizza ora questa strategia di simulazione. Per $i = 1, 2, \ldots, d$, sia n_i il numero di elementi a profondità i nel circuito. Pertanto

$$\sum_{i=1}^{d} n_i = n \ .$$

Si considerino gli elementi combinatori a profondità i. Raggruppandoli in $\lceil n_i/p \rceil$ gruppi, dove i primi $\lceil n_i/p \rceil$ gruppi hanno p elementi ognuno e gli elementi avanzati, se ve ne sono, sono nell'ultimo gruppo, la PRAM può simulare in tempo $O(\lceil n_i/p \rceil)$ il calcolo eseguito da questi elementi combinatori. Il tempo di simulazione totale è perciò dell'ordine di

$$\sum_{i=1}^{d} \left\lceil \frac{n_i}{p} \right\rceil \le \sum_{i=1}^{d} \left(\frac{n_i}{p} + 1 \right)$$
$$= \frac{n}{p} + d \ . \qquad \blacksquare$$

Il Teorema di Brent può essere esteso alla simulazione EREW quando un circuito combinatorio abbia $O(1)$ collegamenti di uscita per ogni elemento combinatorio.

Corollario 30.3

Qualsiasi circuito combinatorio di profondità d e dimensione n con un numero limitato di collegamenti di ingresso e di uscita può essere simulato su una PRAM EREW con p processori in tempo $O(n/p + d)$.

Dimostrazione. Si esegue una simulazione simile a quella della dimostrazione del Teorema di Brent. L'unica differenza è nella simulazione dei fili, per la quale il Teorema 30.2 richiede letture concorrenti. Per la simulazione EREW, l'output calcolato da un elemento combinatorio non è letto direttamente dai processori che richiedono il suo valore. Esso è invece copiato dal processore che simula l'elemento sugli $O(1)$ input che lo richiedono. I processori che necessitano del valore possono poi leggerlo senza interferire l'uno con l'altro. ■

Questa strategia di simulazione EREW non funziona per elementi con un numero non limitato di collegamenti di uscita, poiché la copia può richiedere più di un tempo costante ad ogni passo. Pertanto, per circuiti che hanno elementi con un numero non limitato di collegamenti di uscita c'è bisogno della potenza delle letture concorrenti. (Il caso di un numero non limitato di collegamenti di ingresso può talvolta essere gestito da una simulazione CRCW se gli elementi combinatori sono sufficientemente semplici. Si veda l'Esercizio 30.3-1.)

Il corollario 30.3 fornisce un veloce algoritmo di ordinamento EREW. Come spiegato nelle note al Capitolo 28, la rete di ordinamento AKS di profondità $O(\lg n)$ può ordinare n numeri usando $O(n \lg n)$ confrontatori. Poiché i confrontatori hanno un numero limitato di collegamenti di ingresso, vi è un algoritmo EREW per ordinare n numeri in tempo $O(\lg n)$ usando n processori. (Si è usato questo risultato nel Teorema 30.1 per mostrare che una PRAM EREW può simulare una PRAM CRCW con un rallentamento al più logaritmico.) Sfortunatamente, le costanti nascoste nella notazione O sono così grandi che questo algoritmo di ordinamento è di interesse puramente teorico. Tuttavia sono stati scoperti algoritmi di ordinamento EREW più pratici; da segnalare l'algoritmo parallelo di ordinamento merge-sort di Cole [46].

Si supponga ora di avere un algoritmo PRAM che usa al più p processori, ma di avere una PRAM con solo p' processori con $p' < p$. Si vorrebbe eseguire un algoritmo per p processori sulla PRAM più piccola in modo efficiente rispetto al lavoro. Usando l'idea nella dimostrazione del Teorema di Brent, è possibile stabilire una condizione per quando ciò è possibile.

Teorema 30.4

Se un algoritmo PRAM A per p processori richiede tempo t, allora per qualsiasi $p' < p$ vi è un algoritmo PRAM A' per p' processori che risolve lo stesso problema in tempo $O(pt/p')$.

Dimostrazione. I passi dell'algoritmo A siano numerati $1, 2, \ldots, t$. L'algoritmo A' simula l'esecuzione di ogni passo $i = 1, 2, \ldots, t$ in tempo $O(\lceil p/p' \rceil)$. Vi sono t passi e così l'intera simulazione richiede tempo $O(\lceil p/p' \rceil t) = O(pt/p')$, poiché $p' < p$. ■

Il lavoro eseguito dall'algoritmo A è pt e il lavoro eseguito dall'algoritmo A' è $(pt/p')p' = pt$; la simulazione è dunque efficiente rispetto al lavoro. Di conseguenza, se l'algoritmo A è efficiente rispetto al lavoro, lo è anche l'algoritmo A'.

Quando si sviluppano algoritmi efficienti rispetto al lavoro per un problema, non c'è necessariamente bisogno di creare algoritmi diversi per ogni diverso numero di processori. Per esempio, si supponga di poter dimostrare un limite inferiore stretto t sul tempo di esecuzione di qualunque algoritmo parallelo, indipendentemente dal numero di processori, che risolve un dato problema, e si supponga inoltre che il miglior algoritmo seriale per il problema faccia un lavoro w. Allora, è solamente necessario sviluppare un algoritmo efficiente rispetto al lavoro per il problema che usi $p = \Theta(w/t)$ processori per ottenere un algoritmo efficiente rispetto al lavoro per tutti i numeri di processori per i quali è possibile un algoritmo efficiente rispetto al lavoro. Per $p' = o(p)$, il Teorema 30.4 garantisce l'esistenza di un algoritmo efficiente rispetto al lavoro. Per $p' = \omega(p)$, non esistono algoritmi efficienti rispetto al lavoro, poiché se t è un limite inferiore sul tempo per qualunque algoritmo parallelo, $p't = \omega(pt) = \omega(w)$.

Esercizi

30.3-1 Si dimostri un risultato analogo al Teorema di Brent per una simulazione CRCW del circuito combinatorio booleano avente porte AND e OR con un numero illimitato di collegamenti di ingresso. (*Suggerimento:* sia la "dimensione" il numero totale di input sulle porte del circuito.)

30.3-2 Si mostri come un calcolo parallelo dei prefissi su n valori memorizzati in un array di memoria possa essere realizzato in tempo $O(\lg n)$ su una PRAM EREW usando $O(n/\lg n)$ processori. Perché questo risultato non si estende immediatamente a una lista di n valori?

30.3-3 Si mostri come moltiplicare una matrice A $n \times n$ per un vettore b di n elementi in tempo $O(\lg n)$ con un algoritmo EREW efficiente rispetto al lavoro. (*Suggerimento:* si costruisca un circuito combinatorio per il problema.)

30.3-4 Si dia un algoritmo CRCW che usando n^2 processori moltiplichi due matrici $n \times n$. L'algoritmo dovrebbe essere efficiente rispetto al lavoro rispetto al normale algoritmo seriale di tempo $\Theta(n^3)$ per moltiplicare matrici. Si può progettare un algoritmo EREW?

30.3-5 Alcuni modelli paralleli permettono che i processori diventino inattivi, così che il numero di processori che stanno lavorando vari ad ogni passo. Si definisca il lavoro in questo modello come il numero totale di passi eseguiti dai processori attivi per un algoritmo. Si mostri che qualunque algoritmo CRCW che esegue un lavoro w e impiega tempo t può essere eseguito su una PRAM EREW con p processori in tempo $O((w/p + t)\lg p)$. (*Suggerimento:* la parte più ardua è l'assegnamento dei processori attivi *mentre* il calcolo sta procedendo.)

* 30.4 Calcolo parallelo dei prefissi efficiente rispetto al lavoro

Nel paragrafo 30.1.2, si è esaminato un algoritmo EREW di tempo $O(\lg n)$, LIST-RANK, che può eseguire un calcolo dei prefissi su una lista concatenata di n oggetti. L'algoritmo usa n processori ed esegue lavoro $\Theta(n \lg n)$. Poiché si può facilmente eseguire un calcolo dei prefissi in tempo $\Theta(n)$ su una macchina seriale, LIST-RANK non è efficiente rispetto al lavoro.

Questo paragrafo presenta un algoritmo randomizzato EREW per il calcolo dei prefissi che è efficiente rispetto al lavoro. L'algoritmo usa $\Theta(n \lg n)$ processori e impiega tempo $O(\lg n)$ con alta probabilità. Pertanto è efficiente rispetto al lavoro con alta probabilità. Inoltre, dal Teorema 30.4, da questo algoritmo si ottengono immediatamente algoritmi efficienti rispetto al lavoro per qualunque numero $p = O(n/\lg n)$ di processori.

Calcolo parallelo ricorsivo dei prefissi

L'algoritmo dei prefissi parallelo randomizzato RANDOMIZED-LIST-PREFIX opera su una lista concatenata di n oggetti usando $p = \Theta(n \lg n)$ processori. Durante l'esecuzione dell'algoritmo ogni processore è responsabile di $n/p = \Theta(\lg n)$ oggetti della lista originaria. Gli oggetti sono assegnati ai processori in modo arbitrario (non necessariamente contiguo) prima che la ricorsione cominci e la "proprietà" sugli oggetti non cambia mai. Per comodità, si assume che la lista sia bidirezionale perché la concatenazione doppia di una singola lista richiede tempo $O(1)$.

Figura 30.9 *L'algoritmo parallelo ricorsivo randomizzato efficiente rispetto al lavoro* RANDOMIZED-LIST-PREFIX *per eseguire il calcolo dei prefissi su una lista concatenata di $n = 9$ oggetti.* **(a)-(b)** *Un insieme di oggetti non adiacenti (in nero) sono selezionati per l'eliminazione. Il valore in ogni oggetto nero è usato per aggiornare il valore del prossimo oggetto nella lista, quindi l'oggetto nero è eliminato. L'algoritmo è chiamato ricorsivamente per il calcolo parallelo dei prefissi sulla lista contratta.* **(c)-(d)** *I valori risultanti sono i valori finali corretti per gli oggetti della lista contratta. Gli oggetti eliminati sono poi reinseriti ed ognuno usa il valore dell'oggetto precedente per calcolare il suo valore finale.*

L'idea di RANDOMIZED-LIST-PREFIX è di eliminare alcuni oggetti della lista, eseguire un calcolo dei prefissi ricorsivo sulla lista risultante e quindi espanderla reinserendo gli oggetti eliminati per ottenere un calcolo dei prefissi sulla lista originaria. La figura 30.9 illustra il processo ricorsivo e la figura 30.10 mostra come si sviluppa la ricorsione. Si mostrerà poco più avanti che ogni livello della ricorsione soddisfa due proprietà:

1. Al più un oggetto di quelli appartenenti a un dato processore è selezionato per l'eliminazione.
2. Due oggetti adiacenti non sono selezionati per l'eliminazione.

Prima di mostrare come selezionare gli oggetti in modo che tali proprietà siano soddisfatte, si esamina in maggior dettaglio come viene eseguito il calcolo dei prefissi. Si supponga che al primo passo della ricorsione, il k-esimo oggetto della lista sia selezionato per l'eliminazione. Questo oggetto contiene il valore $[k, k]$ che è caricato dal $(k+1)$-esimo oggetto nella lista. (Situazioni limite, come quella in cui k è alla fine della lista, possono essere gestite direttamente e non sono descritte.) Quindi il $(k+1)$-esimo oggetto, che contiene il valore $[k+1, k+1]$, calcola e memorizza $[k, k+1] = [k, k] \otimes [k+1, k+1]$. Il k-esimo oggetto è poi eliminato dalla lista.

La procedura RANDOMIZED-LIST-PREFIX chiama quindi sé stessa ricorsivamente per eseguire il calcolo dei prefissi sulla lista "contratta". (La ricorsione termina quando la lista è vuota.)

Figura 30.10 *I livelli di ricorsione di* RANDOMIZED-LIST-PREFIX, *mostrati per $n = 9$ oggetti originari. In ogni livello, gli oggetti neri sono eliminati. La procedura si ripete finché la lista è vuota, quindi gli oggetti eliminati vengono reinseriti.*

L'osservazione chiave è che dopo la terminazione dalla chiamata ricorsiva, ogni oggetto nella lista contratta ha il valore finale corretto di cui necessita per il calcolo parallelo dei prefissi sulla lista originaria. Non rimane che reinserire gli oggetti precedentemente eliminati, come l'oggetto k-esimo, e aggiornare i loro valori.

Dopo che il k-esimo oggetto è inserito, il suo valore prefisso finale può essere calcolato usando il valore nel $(k-1)$-esimo oggetto. Dopo la ricorsione il $(k-1)$-esimo oggetto contiene $[1, k-1]$ e così il k-esimo oggetto – che ancora ha il valore $[k, k]$ – ha solo bisogno del valore $[1, k-1]$ per calcolare $[1, k] = [1, k-1] \otimes [k, k]$.

A causa della proprietà 1, ogni elemento selezionato ha un processore distinto per eseguire il lavoro necessario per inserirlo o estrarlo. La proprietà 2 assicura che non sorga nessuna confusione tra i processori quando inseriscono o estraggono oggetti (si veda l'Esercizio 30.4-1).

Le due proprietà insieme assicurano che ogni passo della ricorsione possa essere realizzato in tempo $O(1)$ in un modello EREW.

Selezione di oggetti da eliminare

Come sono selezionati gli oggetti da eliminare dalla procedura RANDOMIZED-LIST-PREFIX? Devono essere rispettate le due proprietà descritte prima e in più si vuole che il tempo richiesto dalla selezione degli oggetti sia breve (e preferibilmente costante). Inoltre si vorrebbe che il maggior numero di oggetti possibile fosse selezionato contemporaneamente.

Il seguente metodo per la selezione randomizzata soddisfa queste condizioni. Gli oggetti sono selezionati attraverso l'esecuzione, da parte di ogni processore, dei seguenti passi:

1. Il processore prende un oggetto i tra quelli che gli appartengono e che non sia stato precedentemente selezionato.
2. Poi "tira una moneta", scegliendo i valori TESTA e CROCE con la stessa probabilità.
3. Se sceglie TESTA, marca l'oggetto i come selezionato, a meno che $next[i]$ sia stato preso da un altro processore la cui moneta abbia dato come risultato TESTA.

Questo metodo randomizzato richiede solo tempo $O(1)$ per selezionare gli oggetti da eliminare e non richiede accessi concorrenti alla memoria.

Si deve mostrare che questa procedura rispetta le due proprietà descritte prima. Si può vedere facilmente che la proprietà 1 è mantenuta, poiché solo un oggetto è scelto da un processore per un'eventuale selezione. Per vedere che anche la proprietà 2 è rispettata, si supponga per assurdo che due oggetti consecutivi i e $next[i]$ siano selezionati. Questo si verifica solo se entrambi sono stati presi dai loro processori e per entrambi i processori il lancio della moneta ha dato come risultato TESTA. Ma l'oggetto i non è selezionato se la moneta del processore responsabile di $next[i]$ ha dato come risultato TESTA, il che è una contraddizione.

Analisi

Poiché ogni passo ricorsivo di RANDOMIZED-LIST-PREFIX impiega tempo $O(1)$, per analizzare l'algoritmo abbiamo solo bisogno di determinare quanti passi richiede l'eliminazione di tutti gli oggetti nella lista originaria. Ad ogni passo, il processore ha almeno probabilità 1/4 di eliminare l'oggetto i che seleziona. Perché? La moneta mostra TESTA con probabilità 1/2 e la probabilità che non selezioni $next[i]$ o lo selezioni ma la moneta mostri CROCE è almeno 1/2. Poiché i lanci delle due monete sono eventi indipendenti, si possono moltiplicare le loro probabilità, ottenendo che la probabilità con cui il processore elimina l'oggetto che seleziona è almeno 1/4. Poiché ogni processore gestisce $\Theta(\lg n)$ oggetti, il tempo medio perché un processore elimini tutti i suoi oggetti è $\Theta(\lg n)$.

Sfortunatamente questa semplice analisi non mostra che il tempo medio di esecuzione di RANDOMIZED-LIST-PREFIX è $\Theta(\lg n)$. Per esempio, se la maggior parte dei processori eliminano tutti i loro oggetti rapidamente e pochi processori richiedono molto più tempo, il tempo medio perché un processore elimini tutti i suoi oggetti potrebbe ancora essere $\Theta(\lg n)$, ma il tempo di esecuzione dell'algoritmo potrebbe essere maggiore.

Il tempo medio di esecuzione della procedura RANDOMIZED-LIST-PREFIX è proprio $\Theta(\lg n)$, anche se la semplice analisi effettuata non lo mostra. Si proverà che con probabilità almeno $1 - 1/n$, tutti gli oggetti sono eliminati in $c \lg n$ livelli di ricorsione, per qualche costante c. Gli Esercizi 30.4-4 e 30.4-5 richiedono di generalizzare questo argomento per dimostrare il limite $\Theta(\lg n)$ sul tempo di esecuzione medio.

L'analisi che segue è basata sull'osservazione che l'esperimento di un dato processore di eliminare gli oggetti che sceglie può essere visto come una sequenza di prove di Bernoulli (si veda il Capitolo 6). L'esperimento è un successo se l'oggetto è selezionato per l'eliminazione, è un fallimento altrimenti. Poiché si è interessati a mostrare che è piccola la probabilità che siano ottenuti pochi successi, si può supporre che i successi si verifichino con probabilità esattamente 1/4 piuttosto che almeno 1/4. (Si vedano gli Esercizi 6.4-8 e 6.4-9 per una giustificazione formale di tale ipotesi.)

Per semplificare ulteriormente l'analisi, si assume che vi siano esattamente $n/\lg n$ processori, ognuno con $\lg n$ oggetti della lista. Si organizzano $c \lg n$ prove, per qualche costante c che determineremo, e si è interessati all'evento che si verifichino meno di $\lg n$ successi. Dal corollario 6.3, la probabilità che un processore elimini meno di $\lg n$ oggetti nelle $c \lg n$ prove è al più

$$
\begin{aligned}
\Pr\{X < \lg n\} &\le \binom{c \lg n}{\lg n}\left(\frac{3}{4}\right)^{c \lg n - \lg n} \\
&\le \left(\frac{ec \lg n}{\lg n}\right)^{\lg n}\left(\frac{3}{4}\right)^{(c-1)\lg n} \\
&= \left(ec\left(\frac{3}{4}\right)^{c-1}\right)^{\lg n} \\
&\le \left(\frac{1}{4}\right)^{\lg n} \\
&= 1/n^2 ,
\end{aligned}
$$

per $c \ge 20$. (La seconda linea segue dalla disuguaglianza (6.9).) Pertanto, la probabilità che tutti gli oggetti appartenenti a un dato processore non siano stati eliminati dopo $c \lg n$ passi è al più $1/n^2$.

Si desidera ora dare un limite superiore alla probabilità che tutti gli oggetti appartenenti a tutti i processori non siano stati eliminati dopo $c \lg n$ passi. Dalla disuguaglianza di Boole (6.22), questa probabilità è al più la somma delle probabilità che ogni processore non abbia eliminato tutti i suoi oggetti. Poiché vi sono $n/\lg n$ processori e ognuno ha probabilità al più $1/n^2$ di non eliminare tutti i suoi oggetti, la probabilità che qualche processore non abbia eliminato tutti i suoi oggetti è al più

$$\frac{n}{\lg n} \cdot \frac{1}{n^2} \le \frac{1}{n} .$$

Si è pertanto dimostrato che con probabilità almeno $1 - 1/n$, ogni oggetto è estratto dopo $O(\lg n)$ chiamate ricorsive. Poiché ogni chiamata ricorsiva richiede tempo $O(1)$, RANDOMIZED-LIST-PREFIX richiede tempo $O(\lg n)$ con alta probabilità.

La costante $c \ge 20$ nel tempo di esecuzione $c \lg n$ può sembrare un po' grande per essere praticabile. In effetti, questa costante è più un prodotto dell'analisi che un riflesso delle prestazioni dell'algoritmo. In pratica, l'algoritmo tende ad essere veloce. I fattori costanti nell'analisi sono grandi perché l'evento che un processore finisca di eliminare tutti i suoi oggetti della lista dipende dall'evento che un altro processore finisca tutto il suo lavoro. A causa di queste dipendenze, si è usata la disuguaglianza di Boole, che non richiede l'indipendenza ma che fornisce una costante più grande di quella che generalmente si verifica in pratica.

Esercizi

30.4-1 Si mostri con delle figure che cosa può non funzionare in RANDOMIZED-LIST-PREFIX se due oggetti adiacenti della lista sono selezionati per l'eliminazione.

* *30.4-2* Si suggerisca una semplice modifica per far eseguire RANDOMIZED-LIST-PREFIX su una lista di n oggetti in tempo $O(n)$ nel caso peggiore. Si usi la definizione di valore medio per dimostrare che con questa modifica l'algoritmo impiega tempo medio $O(\lg n)$.

* *30.4-3* Si mostri come realizzare RANDOMIZED-LIST-PREFIX in modo che usi spazio al più $O(n/p)$ per ogni processore nel caso peggiore, indipendentemente dal numero dei livelli di ricorsione.

* *30.4-4* Si mostri che per qualsiasi costante $k \ge 1$, RANDOMIZED-LIST-PREFIX impiega tempo $O(\lg n)$ con probabilità almeno $1 - 1/n^k$. Si mostri come k influisca sulla costante del tempo di esecuzione.

* *30.4-5* Usando il risultato dell'Esercizio 30.4-4, si mostri che il tempo medio di esecuzione di RANDOMIZED-LIST-PREFIX è $O(\lg n)$.

30.5 Risoluzione deterministica dei conflitti

Si consideri una situazione in cui due processori richiedano di ottenere l'accesso a un oggetto in modo mutuamente esclusivo. Come possono i processori determinare chi debba ottenere l'accesso per primo? Si desidera evitare la situazione in cui entrambi ottengano l'accesso e la situazione in cui nessuno dei due ottenga l'accesso. Il problema di scegliere uno dei processori è un esempio di ***risoluzione di conflitti***. Tutti conoscono l'imbarazzo e la confusione momentanea che si verifica quando due persone cercano di entrare contemporaneamente dalla stessa porta. Simili problemi sono comuni nel progetto di algoritmi paralleli e soluzioni efficienti sono estremamente utili.

Un metodo per risolvere i conflitti è il lancio della moneta. Su un calcolatore il lancio di una moneta può essere realizzato tramite un generatore di numeri casuali. Nell'esempio dei due processori, entrambi possono lanciare la moneta. Se uno ottiene TESTA e l'altro CROCE, quello che ha ottenuto TESTA ottiene l'accesso. Se entrambi ottengono lo stesso valore riprovano ancora. Con questa strategia il conflitto è risolto in tempo medio costante (si veda l'Esercizio 30.5-1).

Si è vista nel Paragrafo 30.4 l'efficacia di una strategia randomizzata. In RANDOMIZED-LIST-PREFIX, oggetti adiacenti della lista non devono essere selezionati per l'eliminazione ma si dovrebbe selezionare il numero maggiore possibile di oggetti. In mezzo a una lista di oggetti scelti, però, tutti gli oggetti sembrano interessanti allo stesso modo. Come si è visto, la randomizzazione fornisce un modo semplice ed efficace per risolvere il conflitto tra oggetti adiacenti della lista mentre garantisce, con alta probabilità, che molti oggetti vengano selezionati.

In questo paragrafo, si studia un metodo deterministico per risolvere i conflitti. La chiave dell'algoritmo è di impiegare gli indici dei processori o gli indirizzi di memoria piuttosto che il lancio casuale di una moneta. Per esempio, nel caso dei due processori, si può risolvere il conflitto permettendo che acceda per primo il processore di indice più piccolo – chiaramente è un processo che richiede tempo costante.

Si userà la stessa idea, ma in un modello molto più furbo, in un algoritmo per risolvere i conflitti in una lista concatenata di n oggetti. L'obiettivo è di scegliere una frazione costante di oggetti nella lista evitando di prendere quelli adiacenti. Questo algoritmo può essere eseguito con n processori in tempo $O(\lg^* n)$ da un algoritmo EREW deterministico. Poiché $\lg^* n \le 5$ per qualunque $n \le 2^{65536}$, per tutti i fini pratici il valore $\lg^* n$ può essere visto come una piccola costante (si veda il paragrafo 2.2).

L'algoritmo deterministico prevede due parti. La prima calcola una "6-colorazione" delle liste concatenate in tempo $O(\lg^* n)$. La seconda converte la "6-colorazione" in un "insieme indipendente massimale" della lista in tempo $O(1)$. L'insieme indipendente massimale conterrà una frazione costante degli n oggetti della lista e nessuna coppia di elementi adiacenti sarà nell'insieme.

Colorazioni e insiemi indipendenti massimali

Una ***colorazione*** di un grafo non orientato $G = (V, E)$ è una funzione $C : V \to \mathbb{N}$ tale che per tutte le coppie $u, v \in V$, se $C(u) = C(v)$, allora $(u, v) \notin E$: cioè non vi sono vertici adiacenti con lo stesso colore. In una 6-colorazione di una lista concatenata, tutti i colori sono nell'insieme $\{0, 1, 2, 3, 4, 5\}$ e non ci sono vertici consecutivi che abbiano lo stesso colore. In effetti qualunque lista concatenata ha una 2-colorazione, poiché si possono colorare gli oggetti di rango dispari con il colore 0 e quelli di rango pari con il colore 1. Si può determinare una tale colorazione in tempo $O(\lg n)$ usando un calcolo parallelo dei prefissi, ma per molte applicazioni è sufficiente determinare solo una $O(1)$-colorazione. Si mostrerà che una 6-colorazione può essere determinata in tempo $O(\lg^* n)$ senza usare la randomizzazione.

Un ***insieme indipendente*** di un grafo $G = (V, E)$ è un sottoinsieme $V' \subseteq V$ di vertici tale che ogni arco in E sia incidente su al più un vertice in V'. Un ***insieme indipendente massimale***, o ***MIS*** (dall'inglese "maximal independent set"), è un insieme indipendente V' tale che, per tutti i vertici $v \in V - V'$, l'insieme $V' \cup \{v\}$ non è indipendente – ogni vertice non in V' è adiacente a qualche vertice in V'. Non si faccia confusione tra il problema di calcolare un insieme indipendente *massimale* – problema facile – con il problema di calcolare un insieme indipendente *massimo* – problema complesso. Il problema di trovare un insieme indipendente di cardinalità massima in un grafo generale è NP-completo. (Si veda il Capitolo 36 per una discussione della NP-completezza. Il Problema 36-1 considera insiemi indipendenti massimi.)

Per liste di n oggetti, un insieme indipendente massimo (e quindi massimale) può essere determinato in tempo $O(\lg n)$ usando un calcolo parallelo dei prefissi, come per identificare gli oggetti con rango dispari nell'appena menzionata 2-colorazione. Questo metodo seleziona $\lceil n/2 \rceil$ oggetti. Si osservi, però che qualunque insieme indipendente massimale di una lista concatenata contiene almeno $n/3$ oggetti, poiché per ogni tripla di oggetti consecutivi, almeno uno deve essere nell'insieme. Si mostrerà però che, data una $O(1)$-colorazione della lista, un insieme indipendente massimale di una lista può essere determinato in tempo $O(1)$.

Calcolo di una 6-colorazione

L'algoritmo SIX-COLOR calcola una 6-colorazione di una lista. Non si vuole fornire lo pseudocodice dell'algoritmo, ma se ne descriveranno alcuni dettagli. Si assuma che inizialmente ogni oggetto x nella lista concatenata sia associato ad un diverso processore $P(x) \in \{0, 1, \ldots, n-1\}$.

L'idea di SIX-COLOR è di calcolare una sequenza $C_0[x], C_1[x], \ldots, C_m[x]$ di colori per ogni oggetto i nella lista. La colorazione iniziale C_0 è una n-colorazione. Ogni iterazione dell'algoritmo definisce una nuova colorazione C_{k+1} basata sulle precedenti colorazioni C_k, per $k = 0, 1, \ldots, m-1$. La colorazione finale C_m è una 6-colorazione e si proverà che $m = O(\lg^* n)$.

La colorazione iniziale è la banale n-colorazione in cui $C_0[x] = P(x)$. Poiché nessuna coppia di oggetti della lista ha lo stesso colore, nessuna coppia di oggetti adiacenti della lista ha lo stesso colore e così la colorazione è legale. Si noti che ognuno dei colori iniziali può essere descritto con $\lceil \lg n \rceil$ bit, il che significa che può essere memorizzato in una comune parola della macchina.

Le successive colorazioni sono ottenute come segue. La k-esima iterazione, per $k = 0, 1, \ldots, m-1$, comincia con una colorazione C_k e termina con una colorazione C_{k+1} usando un numero inferiore di bit per ogni oggetto, come mostra la prima parte della figura 30.11. Si supponga che all'inizio di una iterazione ogni colore C_k degli oggetti richieda r bit. Si determina il nuovo colore di un oggetto x guardando avanti nella lista il colore di $next[x]$.

Per essere più precisi, si supponga che per ogni oggetto x si abbia $C_k[x] = a$ e $C_k[next[x]] = b$, dove $a = \langle a_{r-1}, a_{r-2}, \ldots, a_0 \rangle$ e $b = \langle b_{r-1}, b_{r-2}, \ldots, b_0 \rangle$ sono colori di r bit. Poiché $C_k[x] \ne C_k[next[x]]$, vi è qualche indice i minimo per cui i bit dei due colori sono diversi: $a_i \ne b_i$. Poiché $0 \le i \le r-1$, si può scrivere i con solo $\lceil \lg r \rceil$ bit: $i = \langle i_{\lceil \lg r \rceil - 1}, i_{\lceil \lg r \rceil - 2}, \ldots, i_0 \rangle$. Si ricolora x con il valore di i concatenato con il bit a_i. Cioè, si assegna

$$\begin{aligned} C_{k+1}[x] &= \langle i, a_i \rangle \\ &= \langle i_{\lceil \lg r \rceil - 1}, i_{\lceil \lg r \rceil - 2}, \ldots, i_0, a_i \rangle. \end{aligned}$$

Il fondo della lista contiene il nuovo colore $\langle 0, a_0 \rangle$. Il numero di bit di ciascun nuovo colore è dunque al più $\lceil \lg r \rceil + 1$.

Si deve mostrare che se ogni iterazione di SIX-COLOR comincia con una colorazione, la nuova "colorazione" che produce è in effetti una colorazione legale. Per far questo, si prova che $C_k[x] \ne C_k[next[x]]$ implica $C_{k+1}[x] \ne C_{k+1}[next[x]]$. Si supponga che $C_k[x] = a$ e $C_k[next[x]] = b$, e che $C_{k+1}[x] = \langle i, a_i \rangle$ e $C_{k+1}[next[x]] = \langle j, b_j \rangle$. Vi sono due casi da considerare. Se $i \ne j$, allora $\langle i, a_i \rangle \ne \langle j, b_j \rangle$ e così i nuovi colori sono diversi. Però se $i = j$, allora $a_i \ne b_i = b_j$ per il metodo di ricolorazione e così i nuovi colori sono ancora una volta diversi. (La situazione in fondo alla lista può essere gestita in modo simile.)

Figura 30.11 *Gli algoritmi* SIX-COLOR *e* LIST-MIS *che risolvono i conflitti in una lista. Insieme, gli algoritmi trovano un insieme grande di oggetti non adiacenti in tempo $O(\lg^* n)$ usando n processori. La lista iniziale di $n = 20$ oggetti è mostrata sulla sinistra, in verticale. Ogni oggetto ha un colore iniziale distinto di 5 bit. Per questi parametri l'algoritmo* SIX-COLOR *ha bisogno solo delle due iterazioni mostrate per ricolorare ogni oggetto con un colore nell'intervallo* {0, 1, 2, 3, 4, 5}. *Gli oggetti bianchi sono messi nel MIS con* LIST-MIS *via via che i colori sono elaborati e gli oggetti neri sono eliminati.*

Il metodo di ricolorazione usato da SIX-COLOR richiede un colore di r bit e lo sostituisce con un colore di $\lceil \lg r \rceil + 1$, il che significa che il numero di bit viene strettamente ridotto finché $r \geq 4$. Quando $r = 3$, due colori possono essere diversi nei bit delle posizioni 0, 1 o 2. Ogni nuovo colore, allora, è $\langle 00 \rangle$, $\langle 01 \rangle$ o $\langle 10 \rangle$ concatenato con 0 oppure 1, lasciando così ancora una volta un numero di 3 bit. Sono usati solo 6 degli 8 possibili valori per numeri di 3 bit, così che SIX-COLOR termina proprio con una 6-colorazione.

Supponendo che ogni processore possa determinare l'indice i appropriato in tempo $O(1)$ ed eseguire un'operazione di traslazione a sinistra in tempo $O(1)$ – operazione comunemente offerta da molte macchine reali – ogni iterazione richiede tempo $O(1)$. La procedura SIX-COLOR è un algoritmo EREW: per ogni oggetto x, il suo processore accede solo a x e a $next[x]$.

Infine, si può vedere perché sono richieste solo $O(\lg^* n)$ iterazioni per portare la n-colorazione iniziale fino a una 6-colorazione. Si è definito $\lg^* n$ come il numero di volte che si deve applicare la funzione logaritmica lg ad n per ridurlo a 1 oppure, denotando con $\lg^{(i)} n$ il numero i di applicazioni successive della funzione lg:

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

Sia r_i il numero di bit nella colorazione appena prima della i-esima iterazione. Si proverà per induzione che se $\lceil \lg^{(i)} n \rceil \geq 2$, allora $r_i \leq \lceil \lg^{(i)} n \rceil + 2$. Inizialmente, si ha $r_1 \leq \lceil \lg n \rceil$. L'$i$-esima iterazione riduce i bit della colorazione fino a $r_{i+1} = \lceil \lg r_i \rceil + 1$. Supponendo che l'ipotesi induttiva valga per r_{i-1}, si ottiene

$$\begin{aligned}
r_i &= \lceil \lg r_{i-1} \rceil + 1 \\
&\leq \lceil \lg(\lceil \lg^{(i-1)} n \rceil + 2) \rceil + 1 \\
&\leq \lceil \lg(\lg^{(i-1)} n + 3) \rceil + 1 \\
&\leq \lceil \lg(2 \lg^{(i-1)} n) \rceil + 1 \\
&= \lceil \lg(\lg^{(i-1)} n) + 1 \rceil + 1 \\
&= \lceil \lg^{(i)} n \rceil + 2 .
\end{aligned}$$

La quarta linea segue dall'ipotesi che $\lceil \lg^{(i)} n \rceil \geq 2$, il che significa che $\lceil \lg^{(i-1)} n \rceil \geq 3$. Perciò dopo $m = \lg^* n$ passi, il numero di bit nella colorazione è $r_m \leq \lceil \lg^{(m)} n \rceil + 2 = 3$, poiché $\lg^{(m)} n \leq 1$ per definizione della funzione $\lg^*$. Pertanto, sarà sufficiente al più un'altra iterazione per produrre una 6-colorazione. Il tempo totale di SIX-COLOR è perciò $O(\lg^* n)$.

Calcolo di un MIS da una 6-colorazione

La colorazione è la parte più complessa della risoluzione dei conflitti. L'algoritmo EREW LIST-MIS usa n processori per trovare un insieme indipendente massimale in tempo $O(c)$ data una c-colorazione di una lista di n oggetti. Quindi, dopo aver calcolato una 6-colorazione di una lista, è possibile trovare un insieme indipendente massimale della lista concatenata in tempo $O(1)$.

L'ultima parte della figura 30.11 illustra l'idea di LIST-MIS. Sia data una c-colorazione C. Associato ad ogni oggetto x esiste un bit $alive[x]$ che dice se x è ancora candidato per l'inclusione nel MIS. Inizialmente, $alive[x]$ = TRUE per tutti gli oggetti x.

L'algoritmo è iterato per ognuno dei c colori. Nell'iterazione per il colore i, ogni processore responsabile di un oggetto x controlla se $C[x] = 1$ e $alive[x] =$ TRUE. Se valgono entrambe le condizioni, allora il processore marca x come appartenente al MIS che si sta costruendo. Tutti gli oggetti adiacenti a quelli aggiunti nel MIS – quelli che lo precedono e seguono immediatamente – hanno il bit *alive* che vale FALSE; essi non possono essere nel MIS perché sono adiacenti ad un oggetto nel MIS. Dopo tutte le c iterazioni, ogni oggetto è stato o "eliminato" – il suo bit *alive* è stato messo a FALSE – oppure inserito nel MIS.

Si deve mostrare che l'insieme risultante è indipendente e massimale. Per vedere che è indipendente, si supponga che due oggetti adiacenti x e $next[x]$ siano inseriti nell'insieme. Poiché sono adiacenti, $C[x] \neq C[next[x]]$ perché C è una colorazione. Senza perdita di generalità, si assuma che $C[x] < C[next[x]]$, così che x sia inserito nell'insieme prima di $next[x]$. Ma allora $alive[next[x]]$ è stato messo a FALSE nel momento in cui gli oggetti di colore $C[next[x]]$ sono stati considerati, e $next[x]$ non avrebbe dovuto essere inserito nell'insieme.

Per vedere che l'insieme è massimale, si supponga che nessuno di tre oggetti consecutivi x, y e z sia stato messo nell'insieme. Il solo modo in cui y avrebbe potuto evitare di essere inserito nell'insieme è quello di essere stato eliminato quando un oggetto adiacente era stato inserito nell'insieme. Poiché, per ipotesi, né x né z sono stati messi nell'insieme, l'oggetto y doveva essere ancora vivo nel momento in cui gli oggetti di colore $C[y]$ venivano elaborati e doveva essere messo nel MIS.

Ogni iterazione di LIST-MIS richiede tempo $O(1)$ su una PRAM. L'algoritmo è EREW poiché ogni oggetto accede a se stesso, al suo predecessore e al suo successore nella lista. Combinando LIST-MIS con SIX-COLOR, si possono risolvere deterministicamente i conflitti in una lista concatenata in tempo $O(\lg^* n)$.

Esercizi

30.5-1 Per l'esempio del conflitto tra i due processori dato all'inizio di questo paragrafo, si mostri che il conflitto è risolto in tempo medio costante.

30.5-2 Data una 6-colorazione di una lista di n oggetti, si mostri come effettuare una 3-colorazione della lista in tempo $O(1)$ usando n processori con una PRAM EREW.

30.5-3 Si supponga che ogni nodo diverso dalla radice in un albero di n nodi abbia un puntatore al padre. Si dia un algoritmo CREW per effettuare una $O(1)$-colorazione dell'albero in tempo $O(\lg^* n)$.

* *30.5-4* Si dia un algoritmo PRAM efficiente per effettuare una $O(1)$-colorazione di un grafo in cui ogni vertice ha grado 3. Si analizzi l'algoritmo.

30.5-5 Un ***insieme k-separante*** di una lista concatenata è un insieme di oggetti (separanti) nella lista tali che non vi siano separanti adiacenti e ci siano al più k oggetti non-separanti tra due separanti. Pertanto, un MIS è un insieme 2-separante. Si mostri come un insieme $O(\lg n)$-separante di una lista di n elementi possa essere determinato in tempo $O(1)$ usando n processori. Si mostri come un insieme $O(\lg\lg n)$-separante possa essere determinato in tempo $O(1)$ sotto le stesse ipotesi.

* *30.5-6* Si mostri come trovare una 6-colorazione di una lista concatenata di n oggetti in tempo $O(\lg(\lg^* n))$. Si assuma che ogni processore possa memorizzare una tabella pre-calcolata di dimensione $O(\lg n)$. (*Suggerimento:* nella procedura SIX-COLOR, da quanti valori dipende il colore finale di un oggetto?)

Problemi

30-1 Calcolo segmentato parallelo dei prefissi

Come un normale calcolo dei prefissi, un ***calcolo segmentato dei prefissi*** è definito in termini di un operatore $\otimes$ binario e associativo. Esso riceve una sequenza di input $\langle x_1, x_2, \ldots, x_n\rangle$ i cui elementi sono presi da un dominio S ed una sequenza ***segmento*** $b = \langle b_1, b_2, \ldots, b_n\rangle$ i cui elementi sono presi dal dominio $\{0, 1\}$ con $b_1 = 1$. Produce una sequenza di output $y = \langle y_1, y_2, \ldots, y_n\rangle$ sul dominio S. I bit di b determinano un partizionamento di x e y in segmenti: un nuovo segmento comincia se $b_i = 1$ e quello attuale continua se $b_i = 0$. Il calcolo segmentato dei prefissi esegue un calcolo dei prefissi indipendente dentro ogni segmento di x per produrre il corrispondente segmento di y. La figura 30.12 illustra un calcolo segmentato dei prefissi usando la somma ordinaria.

a. Si definisca l'operatore $\hat{\otimes}$ sulle coppie ordinate (a, z), $(a', z') \in \{0, 1\} \times S$ come segue:

$$(a, z)\ \hat{\otimes}\ (a', z') = \begin{cases} (a, z \otimes z') & \text{se } a' = 0, \\ (1, z') & \text{se } a' = 1 \end{cases}$$

Si provi che $\hat{\otimes}$ è associativo.

b. Si mostri come effettuare qualunque calcolo segmentato dei prefissi su una lista di n elementi in tempo $O(\lg n)$ su una PRAM EREW.

c. Si descriva un algoritmo EREW di tempo $O(k \lg n)$ per ordinare una lista di n numeri di k bit.

30-2 Algoritmo per trovare il massimo efficiente rispetto al numero di processori

Si desidera trovare il massimo di n numeri su una PRAM CRCW con $p = n$ processori.

a. Si mostri che il problema di trovare il massimo di $m \leq p/2$ numeri può essere ridotto al problema di trovare il massimo di al più m^2/p numeri in tempo $O(1)$ su una PRAM CRCW con $p = n$ processori.

b. Se si comincia con $m = \lfloor p/2 \rfloor$ numeri, quanti numeri rimangono dopo k iterazioni dell'algoritmo della parte (a)?

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 calcolo segmentato dei prefissi con sequenza dei segmenti b, sequenza di input x e sequenza di output y. Vi sono 5 segmenti.*

c. Si mostri che il problema di trovare il massimo di n numeri può essere risolto in tempo $O(\lg \lg n)$ su una PRAM CRCW con $p = n$ processori.

30-3 *Componenti connesse*

In questo problema, si studia un algoritmo CRCW arbitrario per calcolare le componenti connesse di un grafo non orientato $G = (V, E)$ che usa $|V + E|$ processori. La struttura di dati usata è una foresta di insiemi disgiunti (si veda il paragrafo 22.3). Ogni vertice $v \in V$ mantiene un puntatore $p[v]$ al padre. Inizialmente, $p[v] = v$: il vertice punta a sé stesso. Alla fine dell'algoritmo, per ogni coppia di vertici $u, v \in V$, si ha $p[u] = p[v]$ se e solo se $u \rightsquigarrow v$ in G. Durante l'esecuzione dell'algoritmo, i puntatori p formano una foresta di alberi radicati di ***puntatori***. Una ***stella*** è un albero di puntatori in cui $p[u] = p[v]$ per tutti i vertici u e v nell'albero.

L'algoritmo delle componenti connesse suppone che ogni arco $(u, v) \in E$ appaia due volte: una volta come (u, v) e una volta come (v, u). L'algoritmo usa due operazioni di base, HOOK e JUMP, e un sottoprogramma STAR che assegna $star[v]$ = TRUE se v appartiene a una stella.

```
HOOK(G)
STAR(G)
for ogni arco (u,v) ∈ E[G], in parallelo
        do if star[u] e p[u] > p[v]
                then p[p[u]] ← p[v]
STAR(G)
for ogni arco (u,v) ∈ E[G], in parallelo
        do if star[u] e p[u] ≠ p[v]
                then p[p[u]] ← p[v]
```

```
JUMP(G)
for ogni v ∈ V[G], in parallelo
        do p[u] ← p[p[v]]
```

L'algoritmo per le componenti connesse inizialmente esegue HOOK una volta, poi esegue ripetutamente HOOK, JUMP, HOOK, JUMP e così via, finché nessun puntatore viene più cambiato dalla operazione JUMP. (Si noti che prima del primo JUMP sono eseguite due HOOK.)

a. Si dia lo pseudocodice per STAR(G).

b. Si mostri che i puntatori p formano una foresta di alberi radicati, con la radice di un albero che punta a sé stessa. Si mostri che se u e v sono nello stesso albero di puntatori, allora $u \rightsquigarrow v$ in G.

c. Si mostri che l'algoritmo è corretto: termina, e quando termina $p[u] = p[v]$ se e solo se $u \rightsquigarrow v$ in G.

Per analizzare l'algoritmo delle componenti connesse, si esamina una singola componente connessa C che si assume abbia almeno due vertici. Si supponga che ad un certo punto dell'esecuzione dell'algoritmo, C sia composto da un insieme $\{T_i\}$ di alberi di puntatori. Si definisca il potenziale di C come

$$\Phi(C) = \sum_{T_i} \text{height}(T_i) \ .$$

L'obiettivo della nostra analisi è di provare che ogni iterazione di HOOK e JUMP diminuisce $\Phi(C)$ di un fattore costante.

d. Si provi che dopo la HOOK iniziale non vi sono alberi di puntatori di altezza 0 e $\Phi(C) \le |V|$.

e. Si spieghi perché dopo la HOOK iniziale le operazioni HOOK successive non aumentano $\Phi(C)$.

f. Si mostri che dopo ogni operazione HOOK non iniziale nessun albero di puntatori è una stella a meno che l'albero di puntatori contenga tutti i vertici di C.

g. Si spieghi perché, se C non è stato compattato in una singola stella, allora, dopo un'operazione JUMP, $\Phi(C)$ è al più 2/3 del suo valore precedente. Si illustri il caso peggiore.

h. Si concluda che l'algoritmo determina tutte le componenti connesse di G in tempo $O(\lg V)$.

30-4 *Traslazione di un'immagine raster*

Una memorizzazione raster di un'immagine può essere vista come una matrice M di $p \times p$ bit. Un dispositivo hardware rende visibile sullo schermo dell'utente la sottomatrice $n \times n$ in alto a sinistra di M. Una operazione BITBLT (dall'inglese "BLock Transfer of BITs", trasferimento in blocco di bit) è usata per muovere un rettangolo di bit da una posizione a un'altra. In particolare, BITBLT($r_1, c_1, r_2, c_2, nr, nc, *$) assegna

$$M[r_2 + i, c_2 + j] \leftarrow M[r_2 + i, c_2 + j] * M[r_1 + i, c_1 + j]$$

per $i = 0, 1, \ldots, nr - 1$ e $j = 0, 1, \ldots, nc - 1$, dove $*$ è una delle 16 funzioni booleane di due input.

Si vuole traslare l'immagine ($M[i, j] \leftarrow M[j, i]$) nella porzione visibile della memorizzazione raster. Si assume che il costo della copia dei bit sia inferiore a quello di chiamare la primitiva BITBLT e quindi si vuole usare il minor numero possibile di operazioni BITBLT.

Si mostri che qualunque immagine sullo schermo può essere trasposta con $O(\lg n)$ operazioni BITBLT. Si assuma che p sia sufficientemente più grande di n in modo che la porzione non visibile della memorizzazione raster fornisca sufficiente memoria di lavoro. Di quanta ulteriore memoria si ha bisogno? (*Suggerimento:* si usi un approccio divide-et-impera in cui alcune delle BITBLT siano eseguite con AND booleani.)

Note al capitolo

Akl[9], Karp e Ramachandran [118] e Leighton [135] esaminano algoritmo paralleli per problemi combinatori. Varie architetture di macchine parallele sono descritte da Hwang e Briggs [109] e Hwang e DeGroot [110].

La teoria del calcolo parallelo cominciò alla fine degli anni 40 quando J. Von Neumann [38] introdusse un modello ristretto di calcolo parallelo chiamato automa cellulare che è essenzialmente un array bidimensionale di processori a stati finiti interconnessi secondo un modello a rete. Il modello PRAM fu formalizzato nel 1978 da Fortune e Wyllie [73], sebbene molti altri autori avessero precedentemente discusso modelli essenzialmente simili.

Il salto dei puntatori fu introdotto da Willie [204]. Lo studio del calcolo parallelo dei prefissi ebbe origine dal lavoro di Ofman [152] nel contesto della addizione con previsione del riporto. La tecnica del ciclo euleriano è dovuta a Tarjan e Vishkin [191].

Valiant [193] ha fornito relazioni tra tempo e processori richiesti per calcolare il massimo di un insieme di n numeri ed ha mostrato anche che non esiste un algoritmo che richieda tempo $O(1)$ e sia efficiente rispetto al lavoro. Cook, Dwork e Reischuck [50] hanno dimostrato che il problema di calcolare il massimo richiede tempo $\Omega(\lg n)$ su una PRAM CREW. La simulazione di un algoritmo CREW con un algoritmo EREW è dovuta a Vishkin [195].

Il Teorema 30.2 è dovuto a Brent [34]. L'algoritmo randomizzato efficiente rispetto al lavoro per il calcolo dei ranghi è stato inventato da Anderson e Miller [11]. Essi presentano anche un algoritmo deterministico efficiente rispetto al lavoro per lo stesso problema [10]. L'algoritmo per la risoluzione dei conflitti è dovuto a Goldberg e Plotkin [84]; esso è basato su un algoritmo simile che ha lo stesso tempo di esecuzione e che è stato inventato da Cole e Vishkin [47].

Operatori sulle matrici

31

Le operazioni sulle matrici sono molto utilizzate nel calcolo scientifico. Algoritmi efficienti per lavorare con le matrici sono dunque di considerevole interesse pratico. Questo capitolo fornisce una breve introduzione alla teoria e alle operazioni sulle matrici, mettendo in rilievo i problemi della moltiplicazione delle matrici e della risoluzione di sistemi di equazioni lineari.

Dopo l'introduzione, nel paragrafo 31.1, di concetti e notazioni di base sulle matrici, il paragrafo 31.2 presenta l'interessante algoritmo di Strassen per moltiplicare due matrici $n \times n$ in tempo $\Theta(n^{\lg 7}) = O(n^{2.81})$. Il paragrafo 31.3 definisce quasianelli, anelli e campi, chiarendo le ipotesi necessarie per il corretto funzionamento dell'algoritmo di Strassen. Esso contiene anche un algoritmo asintoticamente efficiente per moltiplicare matrici booleane. Il paragrafo 31.4 mostra come risolvere un sistema di equazioni lineari usando fattorizzazioni LUP. Quindi, il paragrafo 31.5 studia le strette relazioni tra il problema di moltiplicare matrici e il problema di invertire una matrice. Infine, nel paragrafo 31.6, si discute l'importante classe di matrici simmetriche definite positive e si mostra come esse possano essere usate per trovare una soluzione di un sistema sovradeterminato di equazioni lineari col metodo dei minimi quadrati.

31.1 Proprietà delle matrici

In questo paragrafo, si presentano alcuni concetti di base della teoria delle matrici e alcune proprietà fondamentali delle matrici, mettendo in rilievo quelle che saranno necessarie nei paragrafi successivi.

Matrici e vettori

Una *matrice* è un array rettangolare di numeri. Per esempio,

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \tag{31.1}$$

è una matrice $A = (a_{ij})$ 2×3, dove per $i = 1, 2$ e $j = 1, 2, 3$, l'elemento della matrice nella riga i e nella colonna j è a_{ij}. Si usano le lettere maiuscole per denotare matrici e le corrispondenti

lettere minuscole con gli indici in basso per denotare i loro elementi. L'insieme di tutte le matrici $m \times n$ i cui elementi siano valori reali è denotato con $\mathbb{R}^{m \times n}$. In generale, l'insieme di matrici $m \times n$ con elementi scelti in un insieme S è denotato con $S^{m \times n}$.

La ***trasposta*** di una matrice A è la matrice A^T ottenuta scambiando le righe con le colonne di A. Per la matrice A dell'equazione (31.1),

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} .$$

Un ***vettore*** è un array monodimensionale di numeri. Per esempio,

$$x = \begin{pmatrix} 2 \\ 3 \\ 5 \end{pmatrix} \tag{31.2}$$

è un vettore di dimensione 3. Si usano le lettere minuscole per denotare i vettori e si denota l'i-esimo elemento di un vettore x di dimensione n con x_i per $i = 0, 1, \ldots, n$. Si considera la forma standard di un vettore come un ***vettore colonna*** equivalente a una matrice $n \times 1$: il corrispondente ***vettore riga*** è ottenuto prendendo la trasposta:
$x^T = (2\ 3\ 5)$.

Il ***vettore unità*** e_i è il vettore il cui i-esimo elemento è 1 e tutti gli altri elementi sono 0. Di solito, la dimensione di un vettore unità è chiara dal contesto.

Una ***matrice nulla*** è una matrice in cui ogni elemento è 0. Tale matrice è spesso denotata con 0, poiché l'ambiguità tra il numero 0 e la matrice di zeri di solito si risolve facilmente dal contesto. Nel caso di una matrice di zeri, anche la dimensione della matrice sarà chiara dal contesto.

Si incontrano frequentemente matrici ***quadrate*** $n \times n$. Alcuni casi speciali di matrici quadrate sono di particolare interesse:

1. Una ***matrice diagonale*** ha $a_{ij} = 0$ per ogni $i \neq j$. Poiché tutti gli elementi esterni alla diagonale sono 0, la matrice può essere specificata dall'elenco degli elementi lungo la diagonale:

$$\mathrm{diag}(a_{11}, a_{22}, \ldots, a_{nn}) = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & a_{nn} \end{pmatrix} .$$

2. La ***matrice identità*** I_n è una matrice diagonale $n \times n$ con 1 lungo la diagonale:

$$I_n = diag(1, 1, \ldots, 1) = \begin{pmatrix} 1 & 0 & \ldots & 0 \\ 0 & 1 & \ddots & 0 \\ \vdots & \vdots & \ldots & \vdots \\ 0 & 0 & \ldots & 1 \end{pmatrix} .$$

Quando I appare senza indice, la sua dimensione può essere derivata dal contesto. L'i-esima colonna di una matrice identità è il vettore unità e_i.

3. Una ***matrice tridiagonale*** T è una matrice per cui $t_{ij} = 0$ se $|i - j| > 1$. Elementi diversi da zero appaiono solo sulla diagonale principale, immediatamente sopra la diagonale principale ($t_{i,i+1}$ per $i = 1, 2, \ldots, n-1$) o immediatamente sotto la diagonale principale ($t_{i+1,i}$ per $i = 1, 2, \ldots, n-1$):

$$T = \begin{pmatrix} t_{11} & t_{12} & 0 & 0 & \ldots & 0 & 0 & 0 \\ t_{21} & t_{22} & t_{23} & 0 & \ldots & 0 & 0 & 0 \\ 0 & t_{32} & t_{33} & t_{34} & \ldots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & t_{n-2,n-2} & t_{n-2,n-1} & 0 \\ 0 & 0 & 0 & 0 & \ldots & t_{n-1,n-2} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & 0 & 0 & 0 & \ldots & 0 & t_{n,n-1} & t_{nn} \end{pmatrix} .$$

4. Una ***matrice triangolare superiore*** U è una matrice per cui $u_{ij} = 0$ se $i > j$. Tutti gli elementi sotto la diagonale sono zeri:

$$U = \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1n} \\ 0 & u_{22} & \ldots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & u_{nn} \end{pmatrix} .$$

Una matrice triangolare superiore è ***triangolare superiore unitaria*** se ha tutti 1 lungo la diagonale.

5. Una ***matrice triangolare inferiore*** L è una matrice per cui $l_{ij} = 0$ se $i < j$. Tutti gli elementi sopra la diagonale sono zeri:

$$L = \begin{pmatrix} l_{11} & 0 & \ldots & 0 \\ l_{21} & l_{22} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \ldots & l_{nn} \end{pmatrix} .$$

Una matrice triangolare inferiore è ***triangolare inferiore unitaria*** se ha tutti 1 lungo la diagonale.

6. Una ***matrice di permutazione*** P ha esattamente un 1 in ogni riga o colonna, 0 altrove. Per esempio una matrice di permutazione è

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} .$$

Una tale matrice è chiamata matrice di permutazione perché la moltiplicazione tra un vettore x e una matrice di permutazione ha l'effetto di permutare (risistemare) gli elementi di x.

7. Una ***matrice simmetrica*** A soddisfa la condizione $A = A^T$. Per esempio,

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

è una matrice simmetrica.

Operazioni sulle matrici

Gli elementi di una matrice o di un vettore sono numeri di un sistema di numeri, come i numeri reali, i numeri complessi, o gli interi modulo un numero primo. Il sistema di numeri definisce come sommare o moltiplicare numeri. Si possono estendere queste definizioni in modo da includere anche l'addizione e la moltiplicazione tra matrici.

Si definisce l'***addizione tra matrici*** come segue. Se $A = (a_{ij})$ e $B = (b_{ij})$ sono matrici $m \times n$, allora la loro matrice somma $C = (c_{ij}) = A + B$ è la matrice $m \times n$ definita da

$$c_{ij} = a_{ij} + b_{ij}$$

per $i = 1, 2, \ldots, m$ e $j = 1, 2, \ldots, n$. Cioè, l'addizione tra matrici è eseguita componente per componente. Una matrice nulla è l'elemento neutro per l'addizione tra matrici:

$$\begin{aligned} A + 0 &= A \\ &= 0 + A . \end{aligned}$$

Se λ è un numero e $A = (a_{ij})$ è una matrice, allora $\lambda A = (\lambda a_{ij})$ è il ***prodotto per uno scalare*** di A, ottenuto moltiplicando ogni elemento di A per λ. Come caso particolare, si definisce la ***negativa*** di una matrice $A = (a_{ij})$ come $-1 \cdot A = -A$, così che il generico elemento di $-A$ sia $-a_{ij}$. Pertanto,

$$\begin{aligned} A + (-A) &= 0 \\ &= (-A) + A . \end{aligned}$$

Data questa definizione, si può definire la ***sottrazione tra matrici*** come l'addizione con la negativa di una matrice: $A - B = A + (-B)$.

Si definisce la ***moltiplicazione tra matrici*** come segue. Si considerino due matrici A e B che siano ***compatibili***, nel senso che il numero di colonne di A è uguale al numero di righe di B. (In generale, se un'espressione contiene una matrice prodotto AB si assume sempre che le matrici A e B siano compatibili.) Se $A = (a_{ij})$ è una matrice $m \times n$ e $B = (b_{jk})$ è una matrice $n \times p$, allora la loro matrice prodotto $C = AB$ è la matrice $m \times p$ $C = (c_{ik})$, dove

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \tag{31.3}$$

per $i = 1, 2, \ldots, m$ e $k = 1, 2, \ldots, p$. La procedura MATRIX-MULTIPLY nel paragrafo 26.1 realizza in modo semplice la moltiplicazione tra matrici secondo l'equazione (31.3), assumendo che le matrici siano quadrate: $m = n = p$. Per moltiplicare matrici $n \times n$, MATRIX-MULTIPLY esegue n^3 moltiplicazioni e $n^2(n-1)$ addizioni, e il suo tempo di esecuzione è $\Theta(n^3)$.

Le matrici hanno molte (ma non tutte) proprietà algebriche tipiche dei numeri.

Le matrici identità sono l'elemento neutro della moltiplicazione tra matrici:

$$I_m A = A I_n = A$$

per qualunque matrice A $m \times n$. Moltiplicando per zero una matrice si ottiene una matrice nulla:

$$A0 = 0 .$$

La moltiplicazione tra matrici è associativa:

$$A(BC) = (AB)C \tag{31.4}$$

per matrici compatibili A, B e C. La moltiplicazione tra matrici è distributiva rispetto all'addizione:

$$\begin{aligned} &A(B + C) = AB + BC, \\ &(B + C)D = BD + CD . \end{aligned} \tag{31.5}$$

La moltiplicazione tra matrici $n \times n$ non è, però, commutativa a meno che $n = 1$.

Per esempio se $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ e $B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$, allora

$$AB = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

e

$$BA = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} .$$

I prodotti matrice-vettore o i prodotti vettore-vettore sono definiti come se il vettore fosse l'equivalente di una matrice $n \times 1$ (o una matrice $1 \times n$, nel caso di un vettore riga). Pertanto, se A è una matrice $m \times n$ e x è un vettore di dimensione n, allora Ax è un vettore di dimensione m. Se x e y sono vettori di dimensione n, allora

$$x^T y = \sum_{i=1}^{n} x_i y_i$$

è un numero (in effetti è una matrice 1×1) chiamato ***prodotto interno*** di x e y. La matrice xy^T è una matrice $n \times n$ Z chiamata ***prodotto esterno*** di x e y, con $z_{ij} = x_i y_j$. La ***norma (euclidea)*** $\|x\|$ di un vettore x di dimensione n è definita da

$$\begin{aligned} \|x\| &= (x_1^2 + x_2^2 + \ldots + x_n^2)^{1/2} \\ &= (x^T x)^{1/2} . \end{aligned}$$

Pertanto, la norma di x è la sua lunghezza nello spazio euclideo n-dimensionale.

Matrici inverse, rango e determinante

L'*inversa* di una matrice A $n \times n$ è una matrice $n \times n$, denotata con A^{-1} (se esiste), tale che $AA^{-1} = I_n = A^{-1}A$. Per esempio,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}.$$

Molte matrici $n \times n$ non hanno inversa. Una matrice senza inversa è chiamata ***non invertibile*** o ***singolare***. Un esempio di matrice singolare non nulla è

$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

Se una matrice ammette inversa è chiamata ***invertibile*** o ***non-singolare***. Le matrici inverse, quando esistono, sono uniche. (Si veda l'Esercizio 31.1-4.) Se A e B sono matrici $n \times n$ invertibili, allora

$$(BA)^{-1} = A^{-1}B^{-1}. \tag{31.6}$$

La trasposta dell'inversa è uguale all'inversa della trasposta:

$$(A^{-1})^{\mathrm{T}} = (A^{\mathrm{T}})^{-1}.$$

I vettori $x_1, x_2, \ldots, x_n$ sono ***linearmente dipendenti*** se esistono dei coefficienti $c_1, c_2, \ldots, c_n$, non tutti nulli, tali che $c_1x_1 + c_2x_2 + \ldots + c_nx_n = 0$. Per esempio, i vettori $x_1 = (1\ 2\ 3)^{\mathrm{T}}$, $x_2 = (2\ 6\ 4)^{\mathrm{T}}$ e $x_3 = (4\ 11\ 9)^{\mathrm{T}}$ sono linearmente dipendenti, poiché $2x_1 + 3x_2 - 2x_3 = 0$. Vettori non linearmente dipendenti sono ***linearmente indipendenti***. Per esempio, le colonne di una matrice identità sono linearmente indipendenti.

Il ***rango di colonna*** di una matrice A $m \times n$ non nulla è la cardinalità del più grande insieme di colonne di A linearmente indipendenti. Analogamente, il ***rango di riga*** di A è la cardinalità del più grande insieme di righe di A linearmente indipendenti. Una proprietà fondamentale di qualunque matrice A è che il suo rango di riga è uguale al suo rango di colonna, così che ci si può riferire semplicemente al ***rango*** di A. Il rango di una matrice $m \times n$ è un intero tra 0 e $\min(m, n)$, estremi inclusi. (Infatti, ad esempio, il rango di una matrice nulla è 0 e il rango di una matrice identità $n \times n$ è n.) Una definizione alternativa, ma equivalente e spesso più utile, è che il rango di una matrice A $m \times n$ non nulla è il più piccolo numero r tale che esistono due matrici B e C di dimensione rispettivamente $m \times r$ e $r \times n$ tali che

$$A = BC.$$

Una matrice quadrata $n \times n$ ha ***rango pieno*** se il suo rango è n. Una proprietà fondamentale dei ranghi è data dal seguente teorema.

Teorema 31.1

Una matrice quadrata $n \times n$ ha rango pieno se e solo se è invertibile. ■

Una matrice $m \times n$ ha ***rango di colonna pieno*** se il suo rango è n.

Un ***vettore nullo*** per una matrice A è un vettore x diverso da zero tale che $Ax = 0$.

Il seguente teorema, la cui dimostrazione è richiesta all'Esercizio 31.1-8, e il suo corollario mettono in relazione le nozioni di rango di colonna e invertibilità con i vettori nulli.

Teorema 31.2

Una matrice A ha rango di colonna pieno se e solo se non ha un vettore nullo. ■

Corollario 31.3

Una matrice quadrata A è non invertibile se e solo se ha un vettore nullo. ■

Il ***minore*** ij-esimo di una matrice A $n \times n$, per $n > 1$, è la matrice $A_{[ij]}$ di dimensione $(n-1) \times (n-1)$ ottenuta cancellando l'i-esima riga e la j-esima colonna di A. Il ***determinante*** di una matrice A $n \times n$ può essere definito ricorsivamente in termini dei suoi minori con

$$\det(A) = \begin{cases} a_{11} & \text{se } n = 1, \\ a_{11}\det(A_{[11]}) - a_{12}\det(A_{[12]}) + \ldots + (-1)^{n+1}a_{1n}\det(A_{[1n]}) & \text{se } n > 1. \end{cases} \tag{31.7}$$

Il termine $(-1)^{i+j}\det(A_{[ij]})$ è conosciuto come il ***cofattore*** dell'elemento a_{ij}.

I seguenti teoremi, le cui prove sono state omesse, esprimono le proprietà fondamentali del determinante.

Teorema 31.4 (Proprietà del determinante)

Il determinante di una matrice quadrata A ha le seguenti proprietà:

- Se qualche riga o qualche colonna di A è zero, allora $\det(A) = 0$
- Il determinante di A è moltiplicato per λ se gli elementi in una riga qualunque (o una colonna qualunque) sono tutti moltiplicati per λ.
- Il determinante di A rimane immutato se gli elementi di una riga (rispettivamente colonna) sono sommati a quelli di un'altra riga (rispettivamente colonna).
- Il determinante di A è uguale al determinante di A^{T}.
- Il determinante di A è moltiplicato per -1 se due righe (rispettivamente colonne) qualunque sono scambiate.

Inoltre, per qualunque coppia di matrici quadrate A e B, si ha

$$\det(AB) = \det(A)\det(B).$$ ■

Teorema 31.5

Una matrice A $n \times n$ non è invertibile se e solo se $\det(A) = 0$. ■

Matrici definite positive

Le matrici definite positive giocano un ruolo importante in molte applicazioni. Una matrice A di dimensione $n \times n$ è ***definita positiva*** se $x^{\mathrm{T}}Ax > 0$ per tutti i vettori $x \neq 0$. Per esempio, la matrice identità è definita positiva poiché per qualunque vettore $x = (x_1\ x_2\ \ldots\ x_n)^{\mathrm{T}}$ diverso da zero si ha

$$
\begin{aligned}
x^{\mathrm{T}} I_n x &= x^{\mathrm{T}} x \\
&= \|x\|^2 \\
&= \sum_{i=1}^{n} x_i^2 \\
&> 0 .
\end{aligned}
$$

Come si vedrà, le matrici che si incontrano nelle applicazioni sono spesso definite positive a causa del seguente teorema.

Teorema 31.6

Per qualunque matrice A con rango di colonna pieno, la matrice $A^{\mathrm{T}}A$ è definita positiva.

Dimostrazione. Si deve mostrare che $x^{\mathrm{T}}(A^{\mathrm{T}}A)x > 0$, per qualunque vettore x diverso da zero. Per qualunque vettore x,

$$
\begin{aligned}
x^{\mathrm{T}}(A^{\mathrm{T}}A)x &= (Ax)^{\mathrm{T}}(Ax) \qquad \text{(dall'Esercizio 31.1-3)} \\
&= \|Ax\|^2 \qquad (31.8) \\
&\geq 0 .
\end{aligned}
$$

Si noti che $\|Ax\|^2$ è proprio la somma dei quadrati degli elementi del vettore Ax. Allora, se $\|Ax\|^2 = 0$, ogni elemento di Ax è 0, che è come dire che $Ax = 0$. Poiché A ha rango di colonna pieno, $Ax = 0$ implica $x = 0$, per il Teorema 31.2. Quindi, $A^{\mathrm{T}}A$ è definita positiva. ■

Altre proprietà delle matrici definite positive saranno esaminate nel paragrafo 31.6.

Esercizi

31.1-1 Si dimostri che il prodotto di due matrici triangolari inferiori è triangolare inferiore. Si dimostri che il determinante di una matrice triangolare (inferiore o superiore) è uguale al prodotto degli elementi sulla sua diagonale. Si dimostri che l'inversa di una matrice triangolare inferiore, se esiste, è triangolare inferiore.

31.1-2 Si dimostri che se P è una matrice di permutazione $n \times n$ e A è una matrice $n \times n$, allora PA può essere ottenuta da A permutando le sue righe e AP può essere ottenuta da A permutando le sue colonne. Si provi che il prodotto di due matrici di permutazione è una matrice di permutazione. Si dimostri che se P è una matrice di permutazione, allora P è invertibile, la sua inversa è P^{T} e P^{T} è una matrice di permutazione.

31.1-3 Si dimostri che $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$ e che $A^{\mathrm{T}}A$ è sempre una matrice simmetrica.

31.1-4 Si dimostri che se B e C sono inverse di A, allora $B = C$.

31.1-5 Siano A e B matrici $n \times n$ tali che $AB = I$. Si dimostri che se A' è ottenuta da A sommando la riga j alla riga i, allora l'inversa B' di A' può essere ottenuta sottraendo la colonna i dalla colonna j di B.

31.1-6 Sia A una matrice $n \times n$ invertibile avente come elementi numeri complessi. Si mostri che ogni elemento di A^{-1} è reale se e solo se ogni elemento di A è reale.

31.1-7 Si mostri che se A è una matrice $n \times n$ simmetrica invertibile, allora A^{-1} è simmetrica. Si mostri che se B è una matrice (compatibile) qualunque, allora BAB^{T} è simmetrica.

31.1-8 Si mostri che una matrice A ha rango di colonna pieno se e solo se $Ax = 0$ implica $x = 0$. (*Suggerimento:* si esprima la dipendenza lineare di una colonna dalle altre come un'equazione matrice-vettore.)

31.1-9 Si dimostri che per qualunque coppia di matrici A e B compatibili, $\mathrm{rank}(AB) \leq \min(\mathrm{rank}(A), \mathrm{rank}(B))$, dove l'uguaglianza vale se A (oppure B) è una matrice quadrata invertibile. (*Suggerimento*: si usi la definizione alternativa del rango di una matrice.)

31.1-10 Dati i numeri $x_0, x_1, \ldots, x_{n-1}$, si dimostri che il determinante della ***matrice di Vandermonde***

$$
V(x_0, x_1, \ldots, x_{n-1}) = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}
$$

è

$$
\det(V(x_0, x_1, \ldots, x_{n-1})) = \prod_{0 \leq j < k \leq n-1} (x_k - x_j)
$$

(*Suggerimento:* si moltiplichi la colonna i per $-x_0$, la si sommi alla colonna $i + 1$ per $i = n - 1, n - 2, \ldots, 1$, e poi si usi l'induzione.)

31.2 Algoritmo di Strassen per la moltiplicazione tra matrici

Questo paragrafo presenta l'importante algoritmo ricorsivo di Strassen per moltiplicare matrici $n \times n$ che impiega tempo $\Theta(n^{\lg 7}) = O(n^{2.81})$. Per n sufficientemente grande, allora, è più veloce del semplice algoritmo per la moltiplicazione tra matrici MATRIX-MULTIPLY del paragrafo 26.1 che richiede tempo $\Theta(n^3)$.

Presentazione dell'algoritmo

L'algoritmo di Strassen può essere visto come un'applicazione della tecnica di progettazione "divide et impera".

Si supponga di voler calcolare il prodotto $C = AB$, dove A, B e C sono matrici $n \times n$. Supponendo che n sia una potenza esatta di 2, si divide ogni matrice A, B e C in quattro matrici $n/2 \times n/2$, riscrivendo l'equazione $C = AB$ come segue:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} . \tag{31.9}$$

(L'Esercizio 31.2-2 tratta la situazione in cui n non sia una potenza esatta di 2.) Per comodità, le sottomatrici di A sono etichettate alfabeticamente da sinistra a destra, mentre quelle di B sono etichettate dall'alto in basso, secondo la modalità con cui è eseguita la moltiplicazione tra matrici. L'equazione (3.19) corrisponde alle quattro equazioni

$$r = ae + bf , \tag{31.10}$$

$$s = ag + bh , \tag{31.11}$$

$$t = ce + df , \tag{31.12}$$

$$u = cg + dh . \tag{31.13}$$

Ognuna di queste quattro equazioni specifica due moltiplicazioni di matrici $n/2 \times n/2$ e l'addizione dei loro prodotti. Usando queste equazioni per definire una semplice strategia divide-et-impera, si deriva la seguente ricorrenza per il tempo $T(n)$ per moltiplicare due matrici $n \times n$:

$$T(n) = 8T(n/2) + \Theta(n^2) \tag{31.14}$$

Sfortunatamente la ricorrenza (31.14) ha soluzione $T(n) = \Theta(n^3)$, pertanto questo metodo non è più veloce di quello ordinario.

Strassen scoprì un approccio ricorsivo diverso che richiede solo 7 moltiplicazioni ricorsive di matrici $n/2 \times n/2$ e $\Theta(n^2)$ somme e sottrazioni scalari, portando alla ricorrenza

$$\begin{aligned} T(n) &= 7T(n/2) + \Theta(n^2) \\ &= \Theta(n^{\lg 7}) \\ &= O(n^{2.81}) . \end{aligned} \tag{31.15}$$

Il metodo di Strassen prevede quattro passi:

1. Dividere le matrici di input A e B in sottomatrici $n/2 \times n/2$, come nell'equazione (31.9).
2. Usando $\Theta(n^2)$ addizioni e sottrazioni scalari, calcolare 14 sottomatrici $A_1, B_1, A_2, B_2, \ldots, A_7, B_7$ di dimensione $n/2 \times n/2$.
3. Calcolare ricorsivamente i prodotti delle sette matrici $P_i = A_i B_i$ per $i = 1, 2, \ldots, 7$.
4. Calcolare le sottomatrici richieste r, s, t, u della matrice risultato C aggiungendo e/o sottraendo varie combinazioni delle matrici P_i, usando soltanto $\Theta(n^2)$ addizioni e sottrazioni scalari.

Una tale procedura soddisfa la ricorrenza (31.15). Ciò che rimane da fare ora è specificare i dettagli tralasciati.

Determinazione del prodotto delle sottomatrici

Non è chiaro come Strassen scoprì esattamente quei prodotti di sottomatrici che sono la chiave per rendere veloce il suo algoritmo. In questo paragrafo, si ricostruisce un plausibile metodo di scoperta.

Si fa l'ipotesi che ogni matrice prodotto P_i possa essere scritta nella forma

$$\begin{aligned} P_i &= A_i B_i \\ &= (\alpha_{i1}a + \alpha_{i2}b + \alpha_{i3}c + \alpha_{i4}d) \cdot (\beta_{i1}e + \beta_{i2}f + \beta_{i3}g + \beta_{i4}h) , \end{aligned} \tag{31.16}$$

dove i coefficienti α_{ij}, β_{ij} sono tutti presi dall'insieme $\{-1, 0, 1\}$. Cioè si suppone che ogni prodotto sia calcolato aggiungendo e sottraendo alcune delle sottomatrici di A, aggiungendo e sottraendo alcune delle sottomatrici di B, quindi moltiplicando i due risultati. Anche se sono possibili strategie più generali, questa semplice strategia funziona.

Se si formano tutti i prodotti in questo modo, allora si può usare questo metodo ricorsivamente senza fare ipotesi sulla commutatività della moltiplicazione poiché ogni prodotto ha tutte le sottomatrici di A sulla sinistra e tutte le sottomatrici di B sulla destra. Questa proprietà è essenziale per l'applicazione ricorsiva di questo metodo poiché la moltiplicazione tra matrici non è commutativa.

Per comodità, si useranno matrici 4×4 per rappresentare combinazioni lineari di prodotti di sottomatrici, dove ogni prodotto combina una sottomatrice di A con una sottomatrice di B come nell'equazione (31.16). Per esempio, si può riscrivere l'equazione (31.10) come

$$\begin{aligned} r &= ae + bf \\ &= (a \quad b \quad c \quad d) \begin{pmatrix} +1 & 0 & 0 & 0 \\ 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} \\ &= \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array} . \end{aligned}$$

L'ultima espressione usa una notazione abbreviata in cui il simbolo "+" rappresenta +1, il simbolo "·" rappresenta 0, e il simbolo "−" rappresenta "−1". (D'ora in poi si ometteranno le etichette su righe e colonne.) Usando questa notazione, si hanno le seguenti equazioni per le altre sottomatrici della matrice risultato C:

$$s = ag + bh$$
$$= \begin{pmatrix} \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$t = ce + df$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \end{pmatrix},$$

$$u = cg + dh$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Si comincia la ricerca di un algoritmo di moltiplicazione più veloce osservando che la sottomatrice s può essere calcolata come $s = P_1 + P_2$, dove P_1 e P_2 sono calcolate usando una moltiplicazione di matrici per ciascuna:

$$P_1 = A_1 B_1$$
$$= a \cdot (g - h)$$
$$= ag - ah$$
$$= \begin{pmatrix} \cdot & \cdot & + & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix},$$

$$P_2 = A_2 B_2$$
$$= (a + b) \cdot h$$
$$= ah - bh$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}.$$

La matrice t può essere calcolata in modo analogo come $t = P_3 + P_4$, dove

$$P_3 = A_3 B_3$$
$$= (c + d) \cdot e$$
$$= ce + de$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & \cdot \end{pmatrix}$$

e

$$P_4 = A_4 B_4$$
$$= d \cdot (f - e)$$
$$= df - de$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & + & \cdot & \cdot \end{pmatrix}.$$

Si definisce ***termine essenziale*** ciascuno degli otto termini che compaiono sul lato destro di una delle equazioni (31.10) - (31.13). Sono stati usati 4 prodotti per calcolare le due matrici s e t i cui termini essenziali sono ag, bh, ce e df. Si noti che P_1 calcola il termine essenziale ag, P_2 calcola il termine essenziale bh, P_3 calcola il termine essenziale ce, P_4 calcola il termine essenziale df. Pertanto rimangono da calcolare le due rimanenti sottomatrici r e u, i cui termini essenziali sono i termini della diagonale ae, bf, cg e dh, senza usare più di 3 prodotti aggiuntivi. Si prova ora il nuovo prodotto P_5 per calcolare due termini essenziali in una sola volta:

$$P_5 = A_5 B_5$$
$$= (a + d) \cdot (e + h)$$
$$= ae + ah + de + dh$$
$$= \begin{pmatrix} + & \cdot & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ + & \cdot & \cdot & + \end{pmatrix}.$$

Oltre a calcolare entrambi i termini essenziali ae e dh, P_5 calcola i termini inessenziali ah e de che devono essere cancellati in qualche modo. Si possono usare P_1 e P_2 per cancellarli, ma così appaiono altri due termini inessenziali:

$$P_5 + P_4 - P_2 = ae + dh + df - bh = \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & - \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \end{pmatrix}.$$

Aggiungendo il prodotto

$$\begin{aligned} P_6 &= A_6 B_6 \\ &= (b-d)\cdot(f+h) \\ &= bf + bh - df - dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & + \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & - & \cdot & - \end{pmatrix}. \end{aligned}$$

però, si ottiene

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ &= ae + bf \\ &= \begin{pmatrix} + & \cdot & \cdot & \cdot \\ \cdot & + & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}. \end{aligned}$$

Si può ottenere u in modo analogo da P_5 usando P_1 e P_3 per spostare i termini inessenziali di P_5 verso una direzione diversa:

$$P_5 + P_1 - P_3 = ae + ag - ce + dh = \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}.$$

Sottraendo l'ulteriore prodotto

$$\begin{aligned} P_7 &= A_7 B_7 \\ &= (a-c)\cdot(e+g) \\ &= ae + ag - ce + cg \\ &= \begin{pmatrix} + & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ - & \cdot & - & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}; \end{aligned}$$

si ottiene

$$\begin{aligned} u &= P_5 + P_1 - P_3 - P_7 \\ &= cg + dh \\ &= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & + & \cdot \\ \cdot & \cdot & \cdot & + \end{pmatrix}. \end{aligned}$$

Le sette sottomatrici $P_1, P_2, \ldots, P_7$ possono così essere usate per calcolare il prodotto $C = AB$, il che completa la descrizione del metodo di Strassen.

Discussione

La grande costante nascosta nel tempo di esecuzione dell'algoritmo di Strassen lo rende impraticabile a meno che le matrici siano grandi (con n maggiore o uguale a 45) e dense (con pochi elementi zero). Per matrici piccole è preferibile l'algoritmo semplice e per matrici grandi e sparse vi sono speciali algoritmi che battono in pratica l'algoritmo di Strassen. Pertanto, il metodo di Strassen è soprattutto di interesse teorico.

Usando tecniche avanzate che vanno oltre lo scopo di questo testo, si possono infatti moltiplicare matrici $n \times n$ con tempi migliori di $\Theta(n^{\lg 7})$. L'attuale miglior limite superiore è approssimativamente $O(n^{2,376})$. Il miglior limite inferiore conosciuto è ovviamente $\Omega(n^2)$ (ovviamente perché si devono riempire n^2 elementi della matrice prodotto). Pertanto, attualmente non si conosce quanto la moltiplicazione tra matrici sia realmente complessa.

L'algoritmo di Strassen non richiede che gli elementi delle matrici siano numeri reali. Ciò che è importante è che il sistema di numeri formi un anello algebrico. Tuttavia, se gli elementi della matrice non formano un anello, talvolta possono essere applicate altre tecniche per consentire l'utilizzo di questo metodo. Queste questioni saranno discusse in modo più approfondito nel prossimo paragrafo.

Esercizi

31.2-1 Si usi l'algoritmo di Strassen per calcolare il prodotto delle matrici

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix}\begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}.$$

Si esplicitino i calcoli.

31.2-2 Come si dovrebbe modificare l'algoritmo di Strassen per moltiplicare matrici $n \times n$ in cui n non sia una potenza esatta di 2? Si mostri che l'algoritmo impiega tempo $\Theta(n^{\lg 7})$.

31.2-3 Qual è il più grande k per cui, se si possono moltiplicare matrici 3×3 usando k moltiplicazioni (senza ipotesi sulla commutatività della moltiplicazione), allora si possono moltiplicare matrici $n \times n$ in tempo $o(n^{\lg 7})$? Quale dovrebbe essere il tempo di esecuzione di questo algoritmo?

31.2-4 V. Pan ha scoperto un modo di moltiplicare matrici 68×68 usando 132464 moltiplicazioni, un modo di moltiplicare matrici 70×70 usando 143640 moltiplicazioni e un modo di moltiplicare matrici 72×72 usando 155424 moltiplicazioni. Quale metodo ha il miglior tempo asintotico di esecuzione quando è usato in un algoritmo divide-et-impera per la moltiplicazione tra matrici? Confrontarlo con il tempo di esecuzione dell'algoritmo di Strassen.

31.2-5 Con quale rapidità si possono moltiplicare una matrice $kn \times n$ per una matrice $n \times nk$ usando l'algoritmo di Strassen come sottoprogramma? Si risponda alla stessa domanda con l'ordine invertito delle matrici di input.

31.2-6 Si mostri come moltiplicare i numeri complessi $a + bi$ e $c + di$ usando solo tre moltiplicazioni reali. L'algoritmo dovrebbe ricevere a, b, c e d come input e produrre separatamente la parte reale $ac - bd$ e la parte immaginaria $ad + bc$.

* 31.3 Sistemi algebrici di numeri e moltiplicazione tra matrici booleane

Le proprietà dell'addizione e della moltiplicazione dipendono dalle proprietà del sistema di numeri sottostante. In questo paragrafo, si definiscono tre differenti tipi di sistemi numerici: quasianelli, anelli e campi. Si può definire la moltiplicazione tra matrici sui quasianelli, e l'algoritmo di Strassen per la moltiplicazione tra matrici sugli anelli. Quindi si presenta un semplice trucco per ridurre la moltiplicazione tra matrici booleane, che è definita su un quasianello che non è un anello, alla moltiplicazione su un anello. Infine, si discute il perché le proprietà di un campo non possono essere impiegate naturalmente per fornire algoritmi migliori per la moltiplicazione tra matrici.

Quasianelli

Sia $(S, \oplus, \odot, \bar{0}, \bar{1})$ un sistema di numeri, dove S è un insieme di elementi, $\oplus$ e $\odot$ sono operazioni binarie su S (le operazioni di addizione e moltiplicazione rispettivamente) e $\bar{0}$ e $\bar{1}$ sono due elementi designati in S. Questo sistema è un ***quasianello*** se soddisfa le seguenti proprietà:

1. $(S, \oplus, \bar{0})$ è un ***monoide***:
   - S è ***chiuso*** rispetto a $\oplus$, cioè $a \oplus b \in S$ per ogni $a, b \in S$.
   - $\oplus$ è ***associativo***, cioè $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ per ogni $a, b, c \in S$.
   - $\bar{0}$ è l'***elemento neutro*** per $\oplus$ cioè $a \oplus \bar{0} = \bar{0} \oplus a = a$ per ogni $a \in S$.

   Analogamente $(S, \odot, \bar{1})$ è un monoide.
2. $\bar{0}$ è un ***annullatore***, cioè, $a \odot \bar{0} = \bar{0} \odot a = \bar{0}$ per ogni $a \in S$.
3. L'operatore $\oplus$ è ***commutativo***, cioè, $a \oplus b = b \oplus a$ per ogni $a, b \in S$.
4. L'operatore $\odot$ è ***distributivo*** rispetto a $\oplus$, cioè $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$ e $(b \oplus c) \odot a = (b \odot a) \oplus (c \odot a)$ per ogni $a, b, c \in S$.

Esempi di quasianelli includono il ***quasianello booleano*** $(\{0, 1\}, \vee, \wedge, 0, 1)$, dove $\vee$ denota l'OR logico e $\wedge$ denota l'AND logico, e il sistema dei numeri naturali $(\mathbf{N}, +, \cdot, 0, 1)$, dove $+$ e $\cdot$ denotano le operazioni di addizione e moltiplicazione usuali. Qualunque semianello chiuso (si veda il paragrafo 26.4) è anche un quasianello; i semianelli chiusi godono anche delle proprietà di idempotenza e somme infinite.

Si possono estendere $\oplus$ e $\odot$ alle matrici come si è già fatto per $+$ e $\cdot$ nel paragrafo 31.1. Denotando la matrice identità $n \times n$ composta di $\bar{0}$ e $\bar{1}$ con $\bar{I}_n$, si ha che la moltiplicazione di matrici è ben definita e il sistema di matrici è esso stesso un quasianello, come stabilisce il seguente teorema.

Teorema 31.7 (Matrici su un quasianello formano un quasianello)

Se $(S, \oplus, \odot, \bar{0}, \bar{1})$ è un quasianello e $n \geq 1$, allora $(S^{n \times n}, \oplus, \odot, \bar{0}, \bar{I}_n)$ è un quasianello.

Dimostrazione. La prova è lasciata come Esercizio 31.3-3. ■

Anelli

La sottrazione non è definita per i quasianelli, ma lo è per un ***anello*** che è un quasianello $(S, \oplus, \odot, \bar{0}, \bar{1})$ che soddisfa l'ulteriore proprietà:

5. Ogni elemento in S ha un ***inverso rispetto all'addizione*** cioè, per ogni $a \in S$, esiste un elemento $b \in S$ tale che $a \oplus b = b \oplus a = \bar{0}$. Tale b è anche chiamato l'***opposto*** di a ed è denotato con $(-a)$.

Dato che l'opposto di qualunque elemento è definito, si può definire la sottrazione con $a - b = a + (-b)$.

Vi sono molti esempi di anelli. Gli interi $(\mathbf{Z}, +, \cdot, 0, 1)$ con le comuni operazioni di addizione e moltiplicazione formano un anello. Gli interi modulo n per qualunque intero $n > 1$ – cioè, $(\mathbf{Z}_n, +, \cdot, 0, 1)$, dove $+$ è l'addizione modulo n e $\cdot$ è la moltiplicazione modulo n – formano un anello. Un altro esempio è l'insieme $\mathbf{R}[x]$ di polinomi di grado finito in x con coefficienti reali sotto le comuni operazioni – cioè, $(\mathbf{R}[x], +, \cdot, 0, 1)$, dove $+$ è l'addizione polinomiale e $\cdot$ è la moltiplicazione polinomiale.

Il seguente corollario mostra che il Teorema 31.7 si estende in modo semplice agli anelli.

Corollario 31.8 (Matrici su un anello formano un anello)

Se $(S, \oplus, \odot, \bar{0}, \bar{1})$ è un anello e $n \geq 1$, allora $(S^{n \times n}, \oplus, \odot, \bar{0}, \bar{I}_n)$ è un anello.

Dimostrazione. La prova è lasciata come Esercizio 31.3-3. ■

Usando questo corollario, si può provare il seguente teorema.

Teorema 31.9

L'algoritmo per la moltiplicazione tra matrici di Strassen funziona su qualunque matrice i cui elementi formino un anello.

Dimostrazione. L'algoritmo di Strassen dipende dalla correttezza dell'algoritmo stesso per matrici 2×2, che richiede soltanto che gli elementi appartengano a un anello. Poiché gli elementi della matrice appartengono a un anello, il corollario 31.8 implica che le matrici stesse formino un anello. Pertanto, per induzione, l'algoritmo di Strassen funziona correttamente ad ogni livello di ricorsione. ■

L'algoritmo di Strassen per la moltiplicazione tra matrici dipende in modo critico dall'esistenza dell'opposto. Di sette prodotti $P_1, P_2, \ldots, P_7$, quattro coinvolgono la differenza tra sottomatrici. Pertanto, l'algoritmo di Strassen non funziona in generale per i quasianelli.

Moltiplicazione tra matrici booleane

L'algoritmo di Strassen non può essere usato direttamente per moltiplicare matrici booleane, poiché il quasianello booleano $(\{0,1\}, \vee, \wedge, 0, 1)$ non è un anello. Vi sono esempi in cui un quasianello è contenuto in un sistema più grande che è un anello. Per esempio, i numeri naturali (un quasianello) sono un sottoinsieme degli interi (un anello), e l'algoritmo di Strassen può allora essere usato per moltiplicare le matrici di numeri naturali se si considera come sistema dei numeri sottostante l'insieme degli interi. Sfortunatamente, il quasianello booleano non può essere esteso a un anello in modo analogo. (Si veda l'Esercizio 31.3-4.)

Il seguente teorema presenta un semplice trucco per ridurre la moltiplicazione tra matrici booleane alla moltiplicazione su un anello. Il problema 31-1 presenta un altro approccio efficiente.

Teorema 31.10

Se $M(n)$ denota il numero di operazioni aritmetiche richieste per moltiplicare matrici $n \times n$ di interi, allora due matrici booleane $n \times n$ possono essere moltiplicate usando $O(M(n))$ operazioni aritmetiche.

Dimostrazione. Siano A e B le due matrici, e sia $C = AB$ nel quasianello booleano, cioè.

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj} \ .$$

Invece di operare sul quasianello booleano, si calcola il prodotto C' sull'anello di interi con l'algoritmo dato di moltiplicazione tra matrici che usa $M(n)$ operazioni aritmetiche. Si ha pertanto

$$c'_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \ .$$

Ogni termine $a_{ik}b_{kj}$ di questa somma è 0 se e solo se $a_{ik} \wedge b_{kj} = 0$ mentre vale 1 se e solo se $a_{ik} \wedge b_{kj} = 1$. Pertanto la somma intera c'_{ij} è 0 se e solo se ogni termine è zero, o equivalentemente se e solo se l'OR booleano dei termini, che è c_{ij}, è 0. Allora, la matrice booleana C può essere ricostruita dalla matrice C' di interi semplicemente confrontando c'_{ij} con 0 con $\Theta(n^2)$ operazioni aritmetiche. Il numero di operazioni aritmetiche per l'intera procedura è allora $O(M(n)) + \Theta(n^2) = O(M(n))$, poiché $M(n) = \Omega(n^2)$. ■

Pertanto, usando l'algoritmo di Strassen, si può eseguire la moltiplicazione tra matrici booleane in tempo $O(n^{\lg 7})$.

Il comune metodo per moltiplicare matrici booleane usa solo variabili booleane. Se si usa questo adattamento dell'algoritmo di Strassen, però, la matrice prodotto finale può avere elementi grandi anche fino ad n, richiedendo quindi una parola della macchina piuttosto che un singolo bit per memorizzare un elemento. Il fatto più preoccupante è che i risultati intermedi, che sono interi, possono diventare ancora più grandi di n. Un metodo per evitare che i risultati intermedi crescano troppo è di eseguire tutti i calcoli modulo $n + 1$. L'Esercizio 31.3-5 richiede di mostrare che lavorando con il modulo $n + 1$ non si influenza la correttezza dell'algoritmo.

Campi

Un anello $(S, \oplus, \odot, \bar{0}, \bar{1})$ è un ***campo*** se soddisfa le due ulteriori proprietà:

6. L'operatore $\odot$ è ***commutativo***, cioè $a \odot b = b \odot a$ per ogni $a, b \in S$.
7. Ogni elemento non nullo in S ha un ***inverso rispetto alla moltiplicazione***, cioè, per ogni $a \in S - \{\bar{0}\}$, esiste un elemento $b \in S$ tale che $a \odot b = b \odot a = \bar{1}$. Tale elemento b è spesso chiamato il ***reciproco*** di a ed è denotato con a^{-1}.

Esempi di campi includono i numeri reali $(\mathbf{R}, +, \cdot, 0, 1)$, i numeri complessi $(\mathbf{C}, +, \cdot, 0, 1)$ e gli interi modulo n $(\mathbf{Z}_n, +, \cdot, 0, 1)$.

Poiché i campi offrono i reciproci degli elementi, è possibile definire la divisione. Essi offrono anche la commutatività. Generalizzando dai quasianelli agli anelli, Strassen migliorò il tempo di esecuzione della moltiplicazione tra matrici. Poiché gli elementi di una matrice possono spesso formare un campo – i numeri reali per esempio – si potrebbe sperare che, usando i campi invece degli anelli in un algoritmo ricorsivo tipo quello di Strassen, si possa ulteriormente migliorare il tempo di esecuzione.

Sembra improbabile che questo approccio sia fruttuoso. Per un algoritmo divide-et-impera ricorsivo che per funzionare si basa sui campi, le matrici di ogni passo della ricorsione devono formare un campo. Sfortunatamente, l'estensione ai campi del Teorema 31.7 e del corollario 31.8 fallisce miseramente. Per $n > 1$, l'insieme di matrici $n \times n$ non forma *mai* un campo, anche se il sistema dei numeri sottostante forma un campo. La moltiplicazione di matrici $n \times n$ non è commutativa e molte matrici $n \times n$ non hanno l'inversa. Quindi eventuali algoritmi migliori per la moltiplicazione tra matrici devono essere basati più probabilmente sulla teoria degli anelli piuttosto che sulla teoria dei campi.

Esercizi

* *31.3-1* L'algoritmo di Strassen funziona sul sistema dei numeri $(\mathbf{Z}[x], +, \cdot, 0, 1)$, dove $\mathbf{Z}[x]$ è l'insieme di tutti i polinomi con coefficienti interi nella variabile x e $+$ e $\cdot$ sono le comuni operazioni polinomiali di addizione e moltiplicazione?

* *31.3-2* Si spieghi perché l'algoritmo di Strassen non funziona su semianelli chiusi (si veda il paragrafo 26.4) o sul quasianello booleano $(\{0, 1\}, \vee, \wedge, 0, 1)$.

* *31.3-3* Si dimostri il Teorema 31.7 e il corollario 31.8.

* *31.3-4* Si mostri che il quasianello booleano $(\{0,1\}, \vee, \wedge, 0, 1)$ non può essere incorporato in un anello. Cioè, si mostri che è impossibile aggiungere un "-1" al quasianello così che la struttura algebrica risultante sia un anello.

31.3-5 Si deduca che se tutti i calcoli nell'algoritmo del Teorema 31.10 sono eseguiti modulo $n + 1$, l'algoritmo funziona ancora correttamente.

31.3-6 Si mostri come determinare efficientemente se un dato grafo non orientato contiene un triangolo (un insieme di tre vertici mutuamente adiacenti).

* *31.3-7* Si mostri che calcolare il prodotto di due matrici booleane $n \times n$ sul quasianello booleano è riducibile a calcolare la chiusura transitiva di un dato grafo orientato di $3n$ vertici.

31.3-8 Si mostri come calcolare la chiusura transitiva di un dato grafo orientato di n vertici in tempo $O(n^{\lg 7} \lg n)$. Si confronti questo risultato con le prestazioni della procedura TRANSITIVE-CLOSURE nel paragrafo 26.2.

31.4 Risoluzione di sistemi di equazioni lineari

La risoluzione di un sistema di equazioni lineari è un problema fondamentale in diverse applicazioni. Un sistema lineare può essere espresso come un'equazione tra matrici in cui ogni elemento della matrice o del vettore appartiene a un campo, tipicamente i numeri reali $\mathbf{R}$. Questo paragrafo discute come risolvere un sistema di equazioni lineari usando un metodo chiamato fattorizzazione LUP.

Si comincia con un insieme – o sistema – di equazioni lineari in n incognite $x_1, x_2, \ldots, x_n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 , \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n . \end{aligned} \tag{31.17}$$

Un insieme di valori per $x_1, x_2, \ldots, x_n$ che soddisfi simultaneamente tutte le equazioni (31.17) è detto ***soluzione*** del sistema. In questo paragrafo si tratta solo il caso in cui vi siano esattamente n equazioni in n incognite.

Si possono comodamente riscrivere le equazioni (31.17) come l'equazione matrice-vettore

$$\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

o, equivalentemente, ponendo $A = (a_{ij})$, $x = (x_i)$ e $b = (b_i)$, come

$$Ax = b . \tag{31.18}$$

Se A è invertibile, essa possiede un'inversa A^{-1} e

$$x = A^{-1}b \tag{31.19}$$

è il vettore soluzione. Si può dimostrare che x è l'unica soluzione all'equazione (31.18) come segue. Se vi fossero due soluzioni, x e x', allora $Ax = Ax' = b$ e

$$\begin{aligned} x &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x' . \end{aligned}$$

In questo paragrafo, ci si occuperà principalmente del caso in cui A sia invertibile o, equivalentemente (dal Teorema 31.1), del caso in cui il rango di A sia uguale al numero n di incognite. Vi sono però altre possibilità che meritano una breve discussione. Se il numero di equazioni è minore del numero n di incognite – o, più in generale, se il rango di A è minore di n – allora il sistema è ***sottodeterminato***. Un sistema sottodeterminato tipicamente ha infinite soluzioni (si veda l'Esercizio 31.4-9), sebbene potrebbe non avere alcuna soluzione se le equazioni sono inconsistenti. Se il numero di equazioni è superiore al numero n di incognite, il sistema è ***sovradeterminato*** e può non esistere alcuna soluzione. La ricerca di buone soluzioni approssimate per sistemi di equazioni lineari sovradeterminati è un problema importante che sarà affrontato nel paragrafo 31.6.

Ritornando al problema di risolvere il sistema $Ax = b$ di n equazioni in n incognite, un approccio è quello di calcolare A^{-1} e poi moltiplicare entrambi i lati per A^{-1}, ottenendo $A^{-1}Ax = A^{-1}b$, oppure $x = A^{-1}b$. Questo approccio soffre in pratica di ***instabilità numerica***: errori di arrotondamento tendono ad accumularsi esageratamente quando è usata la rappresentazione in virgola mobile dei numeri invece dei numeri reali ideali. Fortunatamente vi è un altro approccio – la fattorizzazione LUP – che è numericamente stabile ed ha l'ulteriore vantaggio di essere più veloce di un fattore 3.

Presentazione della fattorizzazione LUP

L'idea della fattorizzazione LUP è di trovare tre matrici $n \times n$ L, U e P tali che

$$PA = LU , \tag{31.20}$$

dove

- L è una matrice triangolare inferiore unitaria,
- U è una matrice triangolare superiore e
- P è una matrice di permutazione.

Le matrici L, U e P che soddisfano l'equazione (31.20) sono chiamate una ***fattorizzazione LUP*** della matrice A. Si mostrerà che ogni matrice invertibile A ha una tale fattorizzazione.

Il vantaggio di calcolare una fattorizzazione LUP per la matrice A è che i sistemi lineari possono essere risolti più velocemente quando siano triangolari, come nel caso di entrambe le matrici L e U. Avendo trovato una fattorizzazione LUP per A, si può risolvere l'equazione (31.18) $Ax = b$

risolvendo solo sistemi lineari triangolari come segue. Moltiplicando entrambi i lati di $Ax = b$ per P si ottiene l'equazione equivalente $PAx = Pb$ che dall'Esercizio 31.1-2 equivale alla permutazione dell'equazione (31.17). Usando la fattorizzazione (31.20) si ottiene

$$LUx = Pb .$$

Si può ora risolvere quest'equazione risolvendo due sistemi lineari triangolari. Si definisca $y = Ux$, dove x è il vettore soluzione richiesto. Prima si risolve il sistema triangolare inferiore

$$Ly = Pb \tag{31.21}$$

per il vettore y incognito con un metodo chiamato "sostituzione in avanti". Dopo aver trovato y, si può risolvere il sistema triangolare superiore.

$$Ux = y \tag{31.22}$$

per il vettore x incognito con un metodo chiamato "sostituzione a ritroso". Il vettore x è la soluzione di $Ax = b$ poiché la matrice di permutazione P è invertibile (Esercizio 31.1-2):

$$\begin{aligned} Ax &= P^{-1}LUx \\ &= P^{-1}Ly \\ &= P^{-1}Pb \\ &= b . \end{aligned}$$

Il prossimo passo sarà di mostrare come funzionano la sostituzione in avanti e a ritroso e poi affrontare il problema di calcolare la fattorizzazione LUP stessa.

Sostituzione in avanti e a ritroso

La *sostituzione in avanti* può risolvere il sistema triangolare inferiore (31.21) in tempo $\Theta(n^2)$, dati L, P e b. Per comodità si presenta la permutazione di P in modo compatto con un array $\pi[1..n]$. Per $i = 1, 2, \ldots, n$, l'elemento $\pi[i]$ indica che $P_{i,\pi[i]} = 1$ e $P_{ij} = 0$ per $j \neq \pi[i]$. Pertanto PA ha $a_{\pi[i]j}$ nella riga i e colonna j e Pb ha $b_{\pi[i]}$ come suo i-esimo elemento. Poiché L è triangolare inferiore unitaria, l'equazione (31.21) può essere riscritta come

$$\begin{aligned} y_1 &= b_{\pi[1]} , \\ l_{21}y_1 + y_2 &= b_{\pi[2]} , \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]} , \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]} . \end{aligned}$$

È abbastanza chiaro che si può trovare y_1 direttamente, poiché la prima equazione fornisce $y_1 = b_{\pi[1]}$. Avendo trovato y_1, lo si può sostituire nella seconda equazione, ottenendo

$$y_2 = b_{\pi[2]} - l_{21}y_1 .$$

Ora, si possono sostituire sia y_1 che y_2 nella terza equazione, ottenendo

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2) .$$

In generale si sostituiscono $y_1, y_2, \ldots, y_{i-1}$ "in avanti" nella i-esima equazione per risolvere y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij}y_j .$$

La *sostituzione a ritroso* è analoga alla sostituzione in avanti. Date U e y, si risolve prima la n-esima equazione e poi si opera a ritroso fino alla prima equazione. Come per la sostituzione in avanti, questo processo impiega tempo $\Theta(n^2)$. Poiché U è triangolare superiore, si può riscrivere il sistema (31.22) come

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1 , \\ u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2 , \\ &\vdots \\ u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2} , \\ u_{n-1,n}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} , \\ u_{nn}x_n &= y_n . \end{aligned}$$

Pertanto, si possono calcolare $x_n, x_{n-1}, \ldots, x_1$ una dopo l'altra come segue:

$$\begin{aligned} x_n &= y_n/u_{nn} , \\ x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} , \\ x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} , \\ &\vdots \end{aligned}$$

oppure, in generale,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij}x_j \right) / u_{ii} .$$

Date P, L, U e b, la procedura LUP-SOLVE trova x combinando le sostituzioni in avanti e a ritroso. La procedura assume che la dimensione n compaia nell'attributo $rows[L]$ e che la matrice di permutazione P sia rappresentata dall'array π.

```
LUP-SOLVE(L,U,π,b)
1  n ← rows[L]
2  for i ← 1 to n
3       do y_i ← b_π[i] − Σ_{j=1}^{i−1} l_ij y_j
4  for i ← n downto 1
5       do x_i ← (y_i − Σ_{j=i+1}^{n} u_ij x_j)/u_ii
6  return x
```

La procedura LUP-SOLVE trova y usando la sostituzione in avanti nelle linee 2-3 e poi trova x usando la sostituzione all'indietro nelle linee 4-5. Poiché vi è un ciclo implicito nelle sommatorie dentro ogni ciclo **for**, il tempo di esecuzione è $\Theta(n^2)$.

Come esempio di questi metodi, si consideri il sistema di equazioni lineari definito da

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 0.1 \\ 12.5 \\ 10.3 \end{pmatrix},$$

dove

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 5 & 4 \\ 5 & 6 & 3 \end{pmatrix},$$

$$b = \begin{pmatrix} 0.1 \\ 12.5 \\ 10.3 \end{pmatrix},$$

e si desideri risolverlo nell'incognita x. La fattorizzazione LUP è

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{pmatrix},$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

(Il lettore può verificare che $PA = LU$.) Usando la sostituzione in avanti, si risolve $Ly = Pb$ rispetto a y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.6 & 1 & 0 \\ 0.2 & 0.571 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 10.3 \\ 12.5 \\ 0.1 \end{pmatrix},$$

ottenendo

$$y = \begin{pmatrix} 10.3 \\ 6.32 \\ -5.569 \end{pmatrix}$$

calcolando prima y_1, poi y_2 e infine y_3. Usando la sostituzione a ritroso, si risolve $Ux = y$ rispetto a x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 1.4 & 2.2 \\ 0 & 0 & -1.856 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 10.3 \\ 6.32 \\ -5.569 \end{pmatrix},$$

ottenendo così la soluzione richiesta

$$x = \begin{pmatrix} 0.5 \\ -0.2 \\ 3.0 \end{pmatrix}$$

calcolando prima x_3, poi x_2 e infine x_1.

Calcolo di una fattorizzazione LU

Si è appena mostrato che se può essere calcolata una fattorizzazione LUP per una matrice A invertibile, possono essere usate le sostituzioni in avanti e a ritroso per risolvere il sistema $Ax = b$ di equazioni lineari. Rimane da mostrare come una fattorizzazione LUP per A possa essere trovata in modo efficiente. Si comincia con il caso in cui A è una matrice invertibile $n \times n$ e P è assente (o equivalentemente $P = I_n$). In questo caso, si deve trovare una scomposizione $A = LU$. Le due matrici L e U si chiamano ***fattorizzazione LU*** di A.

Il processo per la fattorizzazione LU è chiamato ***eliminazione di Gauss***. Si comincia sottraendo i multipli della prima equazione dalle altre equazioni così che la prima variabile sia rimossa da quelle equazioni. Poi, si sottraggono i multipli della seconda equazione dalla terza e dalle successive equazioni così che la prima e la seconda variabile siano rimosse. Si continua questo processo finché il sistema ha una forma triangolare superiore – si arriva, cioè, alla matrice U. La matrice L è composta dai moltiplicatori che causano l'eliminazione delle variabili.

L'algoritmo per realizzare questa strategia è ricorsivo. Si vuole costruire una fattorizzazione LU per una matrice A invertibile $n \times n$. Se $n = 1$, allora la costruzione è terminata, poiché si può scegliere $L = I_1$ e $U = A$. Per $n > 1$, si taglia A in quattro parti:

$$A = \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}$$

dove v è un vettore colonna di dimensione $(n-1)$, w^{T} è un vettore riga di dimensione $(n-1)$, e A' è una matrice $(n-1) \times (n-1)$. Quindi, usando l'algebra delle matrici è facile verificare che A ammette la seguente scomposizione:

$$A = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ v / a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T / a_{11} \end{pmatrix} .$$

Gli 0 nella prima e nella seconda matrice della scomposizione sono rispettivamente vettori riga e colonna di dimensione $n - 1$. Il termine vw^T/a_{11}, formato prendendo il prodotto esterno di v e w e dividendo ogni elemento del risultato per a_{11}, è una matrice $(n-1) \times (n-1)$ che corrisponde in dimensione alla matrice A' da cui è sottratta. La matrice $(n-1) \times (n-1)$ risultante

$$A' = vw^T/a_{11} \tag{31.23}$$

è chiamata il ***complemento di Schur*** di A rispetto ad a_{11}.

Si calcola ora ricorsivamente una fattorizzazione LU del complemento di Schur. Sia

$$A' = vw^T/a_{11} = L'U' ,$$

dove L' è triangolare inferiore unitaria e U' è triangolare superiore. Allora, usando l'algebra delle matrici, si ha

$$A = \begin{pmatrix} 1 & 0 \\ v / a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A' - vw^T / a_{11} \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ v / a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L'U' \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ v / a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & U' \end{pmatrix}$$
$$= LU .$$

fornendo così la fattorizzazione LU. (Si noti che poiché L' è triangolare inferiore unitaria anche L lo è, e poiché U' è triangolare superiore anche U lo è.)

Naturalmente, se $a_{11} = 0$, questo metodo non funziona perché si dividerebbe per 0. Non funziona nemmeno se l'elemento più a sinistra in alto del complemento di Schur $A' = vw^T/a_{11}$ è 0, poiché si dividerà per esso nel prossimo passo di ricorsione. Gli elementi per cui si divide durante la fattorizzazione LU sono chiamati ***perni*** e sono gli elementi sulla diagonale della matrice U. La ragione per cui è inclusa una matrice di permutazione P durante la fattorizzazione LUP è che essa consente di evitare di dividere per elementi nulli. L'uso delle permutazioni per evitare la divisione per 0 (o per numeri piccoli) è chiamata ***operazione perno***.

Una importante classe di matrici per cui la fattorizzazione LU funziona sempre correttamente è la classe di matrici simmetriche definite positive. Tali matrici non richiedono l'operazione perno, così la strategia ricorsiva descritta sopra può essere impiegata senza temere di dividere per 0. Si proverà questo risultato ed altri ancora nel paragrafo 31.6.

Il codice per la fattorizzazione LU di una matrice A segue la strategia ricorsiva, tranne che un ciclo iterativo sostituisce la ricorsione. (Questa trasformazione è un'ottimizzazione standard per una procedura "ricorsiva in coda" – quella per cui l'ultima operazione è una chiamata ricorsiva a se stessa.)

Si assume che la dimensione di A sia contenuta nell'attributo *rows*[A]. Poiché si sa che la matrice di output U ha degli 0 sotto la diagonale e poiché LU-SOLVE non opera su questi elementi, il codice non si preoccupa di riempire quelle posizioni. Analogamente, poiché la matrice di output L ha degli 1 sulla sua diagonale e degli 0 sopra la diagonale, nemmeno queste posizioni saranno riempite. Pertanto il codice calcola soltanto gli elementi significativi di L e U.

```
LU-DECOMPOSITION(A)
1  n ← rows[A]
2  for k ← 1 to n
3      do u_kk ← a_kk
4          for i ← k + 1 to n
5              do l_ik ← a_ik/u_kk          ▷ l_ik contiene v_i
6                 u_ki ← a_ki               ▷ u_ki contiene w_i^T
7          for i ← k + 1 to n
8              do for j ← k + 1 to n
9                  do a_ij ← a_ij − l_ik u_kj
10 return L ed U
```

Il ciclo **for** esterno che comincia dalla linea 2 si ripete una volta per ogni passo ricorsivo. Dentro questo ciclo, il perno è determinato come $u_{kk} = a_{kk}$ alla linea 3. Dentro il ciclo **for** alle linee 4-6 (che non sono eseguite quando $k = n$), i vettori v e w^T sono usati per aggiornare L e U. Gli elementi del vettore v sono determinati alla linea 5, dove v_i è memorizzato in l_{ik}, e gli elementi del vettore w^T sono determinati alla linea 6, dove w_i^T è memorizzato in u_{ki}. Infine, gli elementi del complemento di Schur sono calcolati alle linee 7-9 e memorizzati nella matrice A stessa. Poiché la linea 9 è interna a tre cicli, LU-DECOMPOSITION impiega tempo $\Theta(n^3)$.

La figura 31.1 illustra le operazioni di LU-DECOMPOSITION. Mostra un'ottimizzazione standard della procedura in cui gli elementi significativi di L ed U sono memorizzati "in loco" nella matrice A. Cioè si può stabilire una corrispondenza tra ogni elemento a_{ij} e l_{ij} (se $i > j$) oppure tra a_{ij} e u_{ij} (se $i \le j$) e aggiornare la matrice A così che contenga sia U che L quando la procedura termina. Lo pseudocodice per questa ottimizzazione è ottenuto dallo pseudocodice descritto sostituendo semplicemente ogni riferimento a l o u con a; non è difficile verificare che questa trasformazione continua a mantenere la correttezza.

Calcolo di una fattorizzazione LUP

Nella risoluzione dei sistemi di equazioni lineari $Ax = b$, generalmente, si devono usare come perno gli elementi sulla diagonale di A evitando di dividere per 0. Non solo la divisione per 0 è da evitare, ma anche la divisione per valori molto piccoli anche se A è invertibile, perché potrebbero risultare delle instabilità numeriche nel calcolo. Si cerca dunque di usare come perno valori grandi.

Gli aspetti matematici della fattorizzazione LUP sono simili a quelli della fattorizzazione LU. Si ricorda che, data una matrice A $n \times n$ invertibile, si desidera trovare una matrice di

$$\begin{array}{cccc} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{array} \qquad \begin{array}{c|ccc} \bullet 2 & 3 & 1 & 5 \\ \hline 3 & 4 & 2 & 4 \\ 1 & 16 & 9 & 18 \\ 2 & 4 & 9 & 21 \end{array} \qquad \begin{array}{ccc|c} 2 & 3 & 1 & 5 \\ \hline 3 & \bullet 4 & 2 & 4 \\ 1 & 4 & 1 & 2 \\ 2 & 1 & 7 & 17 \end{array} \qquad \begin{array}{ccc|c} 2 & 3 & 1 & 5 \\ \hline 3 & 4 & 2 & 4 \\ 1 & 4 & \bullet 1 & 2 \\ \hline 2 & 1 & 7 & 3 \end{array}$$

(a) (b) (c) (d)

$$\underset{A}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 6 & 13 & 5 & 19 \\ 2 & 19 & 10 & 23 \\ 4 & 10 & 11 & 31 \end{pmatrix}} = \underset{L}{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 2 & 1 & 7 & 1 \end{pmatrix}} \underset{U}{\begin{pmatrix} 2 & 3 & 1 & 5 \\ 0 & 4 & 2 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 3 \end{pmatrix}}$$

(e)

Figura 31.1 *Il comportamento di* LU-DECOMPOSITION. (a) *La matrice A.* (b) *L'elemento $a_{11} = 2$ in nero è il perno, la colonna in grigio è v/a_{11} e la riga in grigio è w^T. Gli elementi di U calcolati finora sono sopra la linea orizzontale e gli elementi di L sono alla sinistra della linea verticale. La matrice complemento di Schur $A' - vw^T/a_{11}$ si trova in basso a destra.* (c) *Si opera ora sulla matrice complemento di Schur prodotta dalla parte* (b). *L'elemento $a_{22} = 4$ in nero è il perno, e la colonna e la riga in grigio sono rispettivamente v/a_{22} e w^T (nel partizionamento del complemento di Schur). Le linee dividono la matrice negli elementi di U calcolati finora (sopra), gli elementi di L calcolati finora (a sinistra) e la nuova matrice complemento di Schur (in basso a destra).* (d) *Il prossimo passo completa la fattorizzazione. (L'elemento 3 nella nuova matrice complemento di Schur diventa parte di U quando termina la ricorsione.)* (e) *La scomposizione $A = LU$.*

permutazione P, una matrice triangolare inferiore unitaria L e una matrice triangolare superiore U tali che $PA = LU$. Innanzitutto si partiziona la matrice A, come si è fatto per la fattorizzazione LU, si muove un elemento diverso da zero della prima colonna, ad esempio a_{k1}, alla posizione (1,1) della matrice. (Se la prima colonna contiene solo degli 0, allora A è non invertibile perché il suo determinante è 0, dai Teoremi 31.4 e 31.5.) Per mantenere il sistema di equazioni, si scambia la riga 1 con la riga k, il che è equivalente a moltiplicare A per una matrice di permutazione Q a sinistra (Esercizio 31.1-2). Pertanto, si può scrivere QA come

$$QA = \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix},$$

dove $v = (a_{21}, a_{31}, \ldots, a_{n1})^T$, eccetto che a_{11} sostituisce a_{k1}; $w^T = (a_{k2}, a_{k3}, \ldots, a_{kn})$ e A' è una matrice $(n-1) \times (n-1)$. Poiché $a_{k1} \neq 0$, si possono ora eseguire gli stessi passi di algebra lineare come già fatto per la fattorizzazione LU, ma in questo caso si ha la garanzia di non dividere per 0:

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^T \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix}. \end{aligned}$$

Il complemento di Schur $A' - vw^T/a_{k1}$ è invertibile perché altrimenti la seconda matrice nell'ultima equazione avrebbe determinante uguale a 0 e pertanto il determinante della

matrice A sarebbe zero; ma ciò significherebbe che A è non invertibile il che contraddice l'ipotesi che A sia invertibile. Di conseguenza si può trovare induttivamente una fattorizzazione LUP per il complemento di Schur con matrice triangolare inferiore L', matrice triangolare superiore U' e matrice di permutazione P', tale che

$$P'(A' - vw^T/a_{k1}) = L'U'.$$

Si definisce

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q,$$

che è una matrice di permutazione, essendo il prodotto di due matrici di permutazione (Esercizio 31.1 – 2). Si ha ora

$$\begin{aligned} PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\ &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & P'(A' - vw^T/a_{k1}) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^T \\ 0 & U' \end{pmatrix} \\ &= LU, \end{aligned}$$

producendo la fattorizzazione LUP. Poiché L' è triangolare inferiore unitaria, tale è anche L e, poiché U' è triangolare superiore, tale è anche U.

Si noti che in questa derivazione, diversamente da quella per la fattorizzazione LU, sia il vettore colonna v/a_{k1} che il complemento di Schur $A' - vw^T/a_{k1}$ devono essere moltiplicati per la matrice di permutazione P'.

Come per LU-DECOMPOSITION, lo psudocodice per la fattorizzazione LUP sostituisce la ricorsione con un ciclo iterativo. Per migliorare la realizzazione della ricorsione, si mantiene dinamicamente la matrice di permutazione P come un array π, dove $\pi[i] = j$ significa che l'i-esima riga di P contiene un 1 nella colonna j. Inoltre si realizzerà il codice per calcolare L e U "in loco" nella matrice A. Pertanto, quando la procedura termina,

$$a_{ij} = \begin{cases} l_{ij} & \text{se } i > j, \\ u_{ij} & \text{se } i \geq j. \end{cases}$$

```
LUP-DECOMPOSITION(A)
1  n ← rows[A]
2  for i ← 1 to n
3        do π[i] ← i
4  for k ← 1 to n
5        do    p ← 0
6              for i ← k to n
7                    do if |a_ik| > p
8                          then  p ← |a_ik|
9                                k' ← i
10             if p = 0
11                   then error "matrice non invertibile"
12             scambia π[k] ↔ π[k']
13             for i ← 1 to n
14                   do scambia a_ki ↔ a_k'i
15             for i ← k + 1 to n
16                   do a_ik ← a_ik/a_kk
17                         for j ← k + 1 to n
18                               do a_ij ← a_ik − a_kj
```

La figura 31.2 illustra come LUP-DECOMPOSITION scomponga una matrice. L'array π è inizializzato alle linee 2-3 per rappresentare la permutazione. Il ciclo **for** esterno che comincia alla linea 4 realizza la ricorsione. Ogni volta che si attraversa il ciclo esterno, le linee 5-9 determinano l'elemento $a_{k'k}$ con valore assoluto più grande tra quelli nella prima colonna corrente (colonna k) della matrice $(n-k+1) \times (n-k+1)$ la cui fattorizzazione LU deve essere trovata. Se tutti gli elementi nella prima colonna sono zero, le linee 10-11 restituiscono il messaggio che la matrice non è invertibile. Per effettuare l'operazione perno, si scambiano $\pi[k']$ con $\pi[k]$ alla linea 12 e si scambiano le righe k e k' di A alle linee 13-14, così che l'elemento perno è a_{kk}. (Sono scambiate intere righe perché nella derivazione del metodo sopra, non solo $A' - vw^T/a_{k1}$ viene moltiplicata per P' ma anche v/a_{k1}.) Infine il complemento di Schur è calcolato alle linee 15-18 praticamente nello stesso modo usato nella LU-DECOMPOSITION, tranne che in questo caso l'algoritmo prevede di operare "in loco". A causa della sua struttura con tre cicli interni, il tempo di esecuzione della LUP-DECOMPOSITION è $\Theta(n^3)$, lo stesso della LU-DECOMPOSITION. Pertanto, l'operazione perno causa una crescita del tempo di esecuzione di al più un fattore costante.

Esercizi

31.4-1 Si risolva l'equazione

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

usando la sostituzione in avanti.

Figura 31.2 *Il comportamento di* LUP-DECOMPOSITION. (a) *La matrice di input A con la permutazione delle righe sulla sinistra. Il primo passo dell'algoritmo determina che l'elemento 5 in nero nella terza riga è il perno per la prima colonna.* (b) *Le righe 1 e 3 sono scambiate e la permutazione è aggiornata. La colonna e la riga in grigio rappresentano v e w^T.* (c) *Il vettore v è sostituito da v/5 e la parte in basso a destra della matrice è aggiornata con il complemento di Schur. Le linee dividono la matrice in tre regioni: elementi di U (sopra), elementi di L (a sinistra) e elementi del complemento di Schur (in basso a destra).* (d)-(f) *Il secondo passo.* (g)-(i) *Il terzo passo. Nessun ulteriore cambiamento occorre nel quarto ed ultimo passo.* (j) *La fattorizzazione LUP, PA = LU.*

31.4-2 Si trovi una fattorizzazione LU della matrice

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix}.$$

31.4-3 Perché il ciclo **for** alla linea 4 di LUP-DECOMPOSITION è eseguito solo fino a $n-1$ mentre il corrispondente ciclo **for** alla linea 2 di LUP-DECOMPOSITION è eseguito fino a n?

31.4-4 Si risolva l'equazione

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

usando la fattorizzazione LUP.

31.4-5 Si descriva la fattorizzazione LUP di una matrice diagonale.

31.4-6 Si descriva la fattorizzazione LUP di una matrice di permutazione A e si dimostri che è unica.

31.4-7 Si mostri che per ogni $n \geq 1$, esistono matrici non invertibili $n \times n$ che hanno fattorizzazioni LU.

* *31.4-8* Si mostri come si può risolvere efficientemente un sistema di equazioni della forma $Ax = b$ sul quasianello booleano $(\{0,1\}, \vee, \wedge, 0, 1)$.

* *31.4-9* Si supponga che A sia una matrice $m \times n$ di reali di rango m, dove $m < n$. Si mostri come trovare un vettore x_0 di dimensione n e una matrice B di dimensione $m \times (n - m)$ e rango $n - m$ tali che ogni vettore della forma $x_0 + By$, per $y \in \mathbf{R}^{n-m}$, sia una soluzione dell'equazione sottodeterminata $Ax = b$.

31.5 Inversione di matrici

Sebbene generalmente l'inversione di matrice non venga praticamente utilizzata per risolvere sistemi di equazioni lineari, preferendo invece usare tecniche numericamente più stabili come la fattorizzazione LUP, è talvolta necessario calcolare l'inversa di una matrice. In questo paragrafo si mostra in che modo la decomposizione LUP possa essere usata per calcolare l'inversa di una matrice. Si discute anche la questione, interessante da un punto di vista teorico, se il calcolo dell'inversa di una matrice possa essere più veloce usando l'algoritmo di Strassen per la moltiplicazione tra matrici. Di fatto, il lavoro originale di Strassen era motivato dal problema di mostrare come un sistema lineare potrebbe essere risolto più velocemente che col metodo usuale.

Calcolo dell'inversa di una matrice da una fattorizzazione LUP

Si supponga di avere una fattorizzazione LUP di una matrice A nella forma di tre matrici L, U e P tali che $PA = LU$. Usando LU-SOLVE, si può risolvere un'equazione della forma $Ax = b$ in tempo $\Theta(n^2)$. Poiché la fattorizzazione LUP dipende da A ma non da b, si può risolvere un secondo sistema di equazioni della forma $Ax = b'$ con un ritardo aggiuntivo dell'ordine di $\Theta(n^2)$. In generale una volta che si sia calcolata la fattorizzazione LUP di A, si possono risolvere, in tempo $\Theta(kn^2)$, k versioni dell'equazione $Ax = b$ che differiscano solo per b.

L'equazione

$$AX = I_n \tag{31.24}$$

può essere vista come un insieme di n equazioni distinte della forma $Ax = b$. Queste equazioni definiscono la matrice X come l'inversa di A. Per essere precisi, si denoti con X_i l'i-esima colonna di X e si ricordi che il vettore unità e_i è l'i-esima colonna di I_n. L'equazione (31.24) può allora essere risolta rispetto a X usando la fattorizzazione LUP per A per risolvere ogni equazione

$$AX_i = e_i$$

separatamente rispetto a X_i. Ogni X_i può essere trovata in tempo $\Theta(n^2)$ e così il calcolo di X dalla fattorizzazione LUP di A richiede tempo $\Theta(n^3)$. Poiché la fattorizzazione LUP di A può essere calcolata in tempo $\Theta(n^3)$, l'inversa A^{-1} di una matrice A può essere determinata in tempo $\Theta(n^3)$.

Moltiplicazione tra matrici e inversione di matrici

Si mostra ora come una maggior velocità nella moltiplicazione tra matrici implichi una maggior velocità per l'inversione di matrici. In effetti, si prova qualcosa di più forte: l'inversione di matrici è equivalente alla moltiplicazione tra matrici, nel senso seguente. Se $M(n)$ denota il tempo per moltiplicare due matrici $n \times n$ ed $I(n)$ denota il tempo per invertire una matrice invertibile $n \times n$, allora $I(n) = \Theta(M(n))$. Si dimostra questo risultato in due parti: prima si mostra che $M(n) = O(I(n))$, che è relativamente facile, quindi si mostra che $I(n) = O(M(n))$.

Teorema 31.11 (La moltiplicazione non è più gravosa dell'inversione)

Se è possibile invertire una matrice $n \times n$ in tempo $I(n)$, dove $I(n) = \Omega(n^2)$ e $I(n)$ soddisfa la condizione di regolarità $I(3n) = O(I(n))$, allora si possono moltiplicare due matrici $n \times n$ in tempo $O(I(n))$.

Dimostrazione. Siano A e B matrici $n \times n$: si vuole calcolare la loro matrice prodotto C. Si definisce la matrice D di dimensione $3n \times 3n$

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

L'inversa di D è

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}.$$

e quindi si può calcolare il prodotto AB prendendo la sottomatrice $n \times n$ di D^{-1} in alto a destra.

Si può costruire la matrice D in tempo $\Theta(n^2) = O(I(n))$ e si può invertire D in tempo $O(I(3n)) = O(I(n))$, dalla condizione di regolarità su $I(n)$. Pertanto si ha

$$M(n) = O(I(n)).$$ ■

Si noti che $I(n)$ soddisfa la condizione di regolarità se $I(n)$ non presenta grandi sbalzi di valore. Per esempio, se $I(n) = \Theta(n^c \lg^d n)$ per qualunque costante $c > 0$, $d \geq 0$, allora $I(n)$ soddisfa la condizione di regolarità.

Riduzione dell'inversione di matrici a moltiplicazione di matrici

La dimostrazione che l'inversione di matrici non è più gravosa della moltiplicazione tra matrici si basa su alcune proprietà delle matrici simmetriche definite positive che saranno dimostrate al paragrafo 31.6.

> ***Teorema 31.12 (L'inversione non è più gravosa della moltiplicazione)***
>
> Se è possibile moltiplicare due matrici $n \times n$ di reali in tempo $M(n)$, dove $M(n) = \Omega(n^2)$ e $M(n) = O(M(n + k))$ per $0 \leq k \leq n$, allora si può calcolare l'inversa di qualunque matrice $n \times n$ invertibile di reali in tempo $O(M(n))$.

Dimostrazione. Si può supporre che n sia una potenza esatta di 2, poiché si ha

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I^k \end{pmatrix}$$

per qualsiasi $k > 0$. Pertanto, scegliendo k tale che $n + k$ sia una potenza di 2, si allarga la matrice fino a una dimensione che è la successiva potenza di 2 e si ottiene la soluzione A^{-1} richiesta dalla soluzione del problema allargato. La condizione di regolarità su $M(n)$ assicura che questo allargamento non causa una crescita del tempo di esecuzione per più di un fattore costante.

Per il momento, si assuma che la matrice A di dimensione $n \times n$ sia simmetrica e definita positiva. Si partiziona A in quattro sottomatrici $n/2 \times n/2$:

$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}, \tag{31.25}$$

Allora, posto

$$S = D - CB^{-1}C^T \tag{31.26}$$

il complemento di Schur di A rispetto a B, si ha

$$A' = \begin{pmatrix} B^{-1} + B^{-1}C^T S^{-1} CB^{-1} & -B^{-1}C^T S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}, \tag{31.27}$$

poiché $AA^{-1} = I_n$, come si può verificare eseguendo la moltiplicazione tra matrici. Le matrici B^{-1} e S^{-1} esistono se A è simmetrica e definita positiva, dai Lemmi 31.13, 31.14 e 31.15 nel paragrafo 31.6, poiché sia B che S sono simmetriche e definite positive. Dall'Esercizio 31.1-3, si ottiene $B^{-1}C^T = (CB^{-1})^T$ e $B^{-1}C^T S^{-1} = (S^{-1}CB^{-1})^T$.

Le equazioni (31.6) e (31.7) possono allora essere usate per specificare un algoritmo ricorsivo che preveda 4 moltiplicazioni di matrici $n/2 \times n/2$:

$C \cdot B^{-1}$,

$(CB^{-1}) \cdot C^T$,

$S^{-1} \cdot (CB^{-1})$,

$(CB^{-1})^T \cdot (S^{-1}CB^{-1})$.

Poiché si possono moltiplicare matrici $n/2 \times n/2$ usando un algoritmo per matrici $n \times n$, l'inversione di matrici simmetriche definite positive può essere eseguita in tempo

$$\begin{aligned} I(n) &\leq 2I(n/2) + 4M(n) + O(n^2) \\ &= 2I(n/2) + O(M(n)) \\ &= O(M(n)) . \end{aligned}$$

Rimane da dimostrare che il tempo di esecuzione asintotico della moltiplicazione tra matrici può essere ottenuto anche per l'inversione di matrici quando A è invertibile ma non simmetrica e definita positiva. L'idea di base è che per qualsiasi matrice invertibile A, la matrice A^TA è simmetrica (dall'Esercizio 31.1-3) e definita positiva (dal Teorema 31.6). Il trucco, allora, è di ridurre il problema di invertire A al problema di invertire A^TA.

La riduzione si basa sull'osservazione che quando A è una matrice invertibile $n \times n$, si ha

$A^{-1} = (A^TA)^{-1}A^T$,

poiché $((A^TA)^{-1}A^T)A = (A^TA)^{-1}(A^TA) = I_n$ e una matrice inversa è unica. Allora, si può calcolare A^{-1} prima moltiplicando A^T per A per ottenere A^TA, quindi invertire la matrice A^TA simmetrica definita positiva usando l'algoritmo divide-et-impera visto precedentemente e infine moltiplicando il risultato per A^T. Ognuno di questi tre passi richiede tempo $O(M(n))$. ■

La dimostrazione del Teorema 31.12 suggerisce un metodo per risolvere l'equazione $Ax = b$ senza eseguire l'operazione perno, se A è invertibile. Si moltiplicano entrambi i lati dell'equazione per A^T, ottenendo $(A^TA)x = A^Tb$. Questa trasformazione non influenza la soluzione x poiché A è invertibile; si può allora scomporre la matrice A^TA simmetrica definita positiva calcolando una fattorizzazione LU. Quindi si usano le sostituzioni in avanti e a ritroso per risolvere rispetto a x con il lato destro A^Tb. Sebbene questo metodo sia teoricamente corretto, la procedura LUP-DECOMPOSITION funziona molto meglio in pratica. La fattorizzazione LUP richiede meno operazioni aritmetiche per un fattore costante e per certi versi ha proprietà numeriche migliori.

Esercizi

31.5-1 Sia $M(n)$ il tempo per moltiplicare matrici $n \times n$ e si denoti con $S(n)$ il tempo richiesto per calcolare il quadrato di una matrice $n \times n$. Si mostri che la moltiplicazione e il quadrato di matrici presentano essenzialmente le stesse difficoltà: $S(n) = \Theta(M(n))$.

31.5-2 Sia $M(n)$ il tempo per moltiplicare matrici $n \times n$ e sia $L(n)$ il tempo richiesto per calcolare la fattorizzazione LUP di una matrice $n \times n$. Si mostri che la moltiplicazione e la fattorizzazione LUP di matrici presentano essenzialmente le stesse difficoltà: $L(n) = \Theta(M(n))$.

31.5-3 Sia $M(n)$ il tempo per moltiplicare matrici $n \times n$ e si denoti con $D(n)$ il tempo richiesto per calcolare il determinante di una matrice $n \times n$. Si mostri che il calcolo

del determinante non è più gravoso della moltiplicazione tra matrici: $D(n) = \Theta(M(n))$.

31.5-4 Sia $M(n)$ il tempo per moltiplicare matrici booleane $n \times n$ e sia $T(n)$ il tempo richiesto per trovare la chiusura transitiva di una matrice booleana $n \times n$. Si mostri che $M(n) = \Theta(T(n))$ e $T(n) = O(M(n)\lg n)$.

31.5-5 L'algoritmo per l'inversione di una matrice basato sul Teorema 31.12 funziona quando gli elementi della matrice sono presi da un campo di interi modulo 2? Giustificare la risposta.

* *31.5-6* Si generalizzi l'algoritmo al Teorema 31.12 per l'inversione di una matrice affinché tratti matrici di numeri complessi e si dimostri che la generalizzazione funziona correttamente. (*Suggerimento:* invece della trasposta di A, si usi la ***trasposta coniugata*** A^* che è ottenuta dalla trasposta di A sostituendo ogni elemento con il suo complesso coniugato. Invece di matrici simmetriche, si considerino le matrici ***hermitiane*** che sono matrici A tali che $A = A^*$.)

31.6 Matrici simmetriche definite positive e metodo dei minimi quadrati

Le matrici simmetriche definite positive presentano molte proprietà interessanti e utili. Per esempio, sono invertibili e la fattorizzazione LU può essere eseguita su di esse senza la preoccupazione di dividere per 0. In questo paragrafo, si dimostreranno alcune importanti proprietà delle matrici simmetriche definite positive e si mostrerà un'applicazione interessante per descrivere una curva in modo approssimato attraverso il metodo dei minimi quadrati.

La prima proprietà che si dimostra è forse quella più importante.

Lemma 31.13

Qualsiasi matrice simmetrica definita positiva è invertibile.

Dimostrazione. Si supponga che una matrice A sia non invertibile. Allora, dal corollario 31.3, esiste un vettore non nullo x tale che $Ax = 0$. Quindi, $x^T Ax = 0$ e A non può essere definita positiva. ■

La dimostrazione che si può eseguire una fattorizzazione LU su una matrice A simmetrica definita positiva senza dividere per 0 è più complessa. Si comincia provando proprietà di certe sottomatrici di A. Si definisce la k-esima ***sottomatrice portante*** di A come la matrice A_k formata dalle prime k righe e dalle prime k colonne di A.

Lemma 31.14

Se A è una matrice simmetrica definita positiva, allora ogni sottomatrice portante di A è simmetrica e definita positiva.

Dimostrazione. Il fatto che ogni sottomatrice portante di A sia simmetrica è ovvio. Per dimostrare che A_k è definita positiva, sia x un vettore colonna non zero di dimensione k, e sia A partizionata come segue:

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}.$$

Allora si ha

$$\begin{aligned} x^T A_k x &= \begin{pmatrix} x^T & 0 \end{pmatrix} \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \begin{pmatrix} x \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} x^T & 0 \end{pmatrix} A \begin{pmatrix} x \\ 0 \end{pmatrix} \\ &> 0 , \end{aligned}$$

poiché A è definita positiva e quindi anche A_k è definita positiva. ■

Si passa ora ad alcune proprietà essenziali del complemento di Schur. Sia A una matrice simmetrica definita positiva e sia A_k una sottomatrice portante di A. Si partiziona A come

$$A = \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix}. \tag{31.28}$$

Allora, il ***complemento di Schur*** di A rispetto ad A_k è definito come

$$S = C - BA_k^{-1}B^T. \tag{31.29}$$

(Dal lemma 31.14, A_k è simmetrica e definita positiva; allora A_k^{-1} esiste per il lemma 31.13 ed S è ben definita.) Si noti che la prima definizione (31.23) del complemento di Schur è consistente con la definizione (31.29) ponendo $k = 1$.

Il prossimo lemma mostra che le matrici complemento di Schur di matrici simmetriche definite positive sono esse stesse matrici simmetriche e definite positive. Questo risultato è stato usato nel Teorema 31.12 e il suo corollario è necessario per dimostrare la correttezza della fattorizzazione LU per matrici simmetriche definite positive.

Lemma 31.15 (Lemma del complemento di Schur)

Se A è una matrice simmetrica definita positiva e A_k è una sottomatrice portante di A, allora la matrice complemento di Schur di A rispetto ad A_k è simmetrica e definita positiva.

Dimostrazione. Dall'Esercizio 31.1-7 segue che S è simmetrica. Rimane da mostrare che S è definita positiva. Si consideri la partizione di A data nell'equazione (31.28).

Per qualunque vettore x non nullo, si ha $x^T Ax > 0$ per ipotesi. Si spezzi x in due sottovettori y e z compatibili con A_k e C, rispettivamente. Poiché A_k^{-1} esiste, si ha

$$x^T A_k x = \begin{pmatrix} y^T & z^T \end{pmatrix} \begin{pmatrix} A_k & B^T \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix}$$

$$= y^T A_k y + y^T B^T z + z^T B y + z^T C z$$
$$= (y + A_k^{-1} B^T z)^T A_k (y + A_k^{-1} B^T z) + z^T (C - B A_k^{-1} B^T) z \ , \qquad (31.30)$$

come per magia. (Si verifichi eseguendo le moltiplicazioni.) Quest'ultima equazione equivale al "completamento del quadrato" della forma quadratica. (Si veda l'Esercizio 31.6-2.)

Poiché $x^T A x > 0$ vale per qualunque vettore x non nullo, si prenda un qualsiasi vettore z non nullo e quindi si scelga $y = -A_k^{-1} B^T z$ che causa l'eliminazione del primo termine dell'equazione (31.30), lasciando

$$z^T (C - A_k^{-1} B^T) z = z^T S z$$

come valore dell'espressione. Per qualunque $z \neq 0$, si ha dunque $z^T S z = x^T A x > 0$, pertanto S è definita positiva. ■

Corollario 31.16

La fattorizzazione LU di una matrice simmetrica definita positiva non provoca mai una divisione per 0.

Dimostrazione. Sia A una matrice simmetrica definita positiva. Si proverà qualcosa di più forte dell'asserzione del corollario: ogni perno è strettamente positivo. Il primo perno è a_{11}. Sia e_1 il primo vettore unitario, da cui si ottiene $a_{11} = e_1^T A e_1 > 0$. Poiché il primo passo della fattorizzazione LU produce il complemento di Schur di A rispetto ad $A_1 = (a_{11})$, il lemma 31.15 implica, per induzione, che tutti i perni siano positivi. ■

Metodo dei minimi quadrati

L'interpolazione di un dato insieme di punti mediante una curva è un'importante applicazione delle matrici simmetriche definite positive. Si supponga che sia dato un insieme di m punti

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m) \ ,$$

e che si sappia che gli y_i sono soggetti a errori di misurazione. Si vorrebbe determinare una funzione $F(x)$ tale che

$$y_i = F(x_i) + \eta_i \ , \qquad (31.31)$$

per $i = 1, 2, \ldots, m$, dove gli errori di approssimazione η_i siano piccoli. La forma della funzione F dipende dal problema che si sta trattando. In questa sede, si assume che abbia la forma di una somma pesata linearmente

$$F(x) = \sum_{j=1}^{n} c_j f_j(x) \ ,$$

dove il numero n di addendi e le specifiche ***funzioni di base*** f_j sono scelte basandosi sulla conoscenza del problema che si sta trattando. Una scelta comune è $f_j(x) = x^{j-1}$, da cui

$$F(x) = c_1 + c_2 x + c_3 x^2 + \ldots + c_n x^{n-1}$$

risulta essere un polinomio di grado $n - 1$ in x.

Scegliendo $n = m$, si può calcolare *in modo esatto* ciascun y_i dell'equazione (31.31). Tuttavia, una funzione F di grado così alto risente pesantemente degli errori di misurazione e generalmente dà miseri risultati quando è usata per prevedere il valore di y per valori di x che non siano stati precedentemente osservati. Di solito è meglio scegliere n significativamente più piccolo di m e sperare che, scegliendo bene i coefficienti c_j, si possa ottenere una funzione F che dia un'approssimazione significativa, ma non necessariamente esatta, nei punti dati. Esistono alcuni principi teorici per scegliere n, ma vanno oltre la portata di questo testo. In ogni caso, una volta scelto n, si arriva ad un insieme sovradeterminato di equazioni che si desidera risolvere nel modo migliore possibile. Si mostra ora come ciò può essere fatto.

Sia

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \ldots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \ldots & f_n(x_2) \\ \vdots & \vdots & \ldots & \vdots \\ f_1(x_m) & f_2(x_m) & \ldots & f_n(x_m) \end{pmatrix}$$

la matrice di valori delle funzioni di base nei punti dati; cioè $a_{ij} = f_j(x_i)$. Sia $c = (c_k)$ il vettore dei coefficienti di dimensione n richiesto. Allora,

$$Ac = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \ldots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \ldots & f_n(x_2) \\ \vdots & \vdots & \ldots & \vdots \\ f_1(x_m) & f_2(x_m) & \ldots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$$
$$= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix}$$

è il vettore di dimensione m di "valori previsti" per y. Pertanto,

$$\eta = Ac - y$$

è il vettore di dimensione m degli ***errori di approssimazione***.

Per minimizzare gli errori di approssimazione, si sceglie di minimizzare la norma del vettore degli errori η, che fornisce una ***soluzione ai minimi quadrati***, dato che

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2} \ .$$

Poiché

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m}\left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right)^2 ,$$

si può minimizzare $\|\eta\|$ derivando $\|\eta\|^2$ rispetto ad ogni c_k e quindi uguagliando il risultato a 0:

$$\frac{d\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2\left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right)a_{ik} = 0 . \tag{31.32}$$

Le n equazioni (31.32) per $k = 1, 2, \ldots, n$ sono equivalenti alla singola equazione matriciale

$$(Ac - y)^{\mathrm{T}}A = 0$$

oppure, equivalentemente (si usi l'Esercizio 31.1-3), a

$$A^{\mathrm{T}}(Ac - y) = 0$$

che implica

$$A^{\mathrm{T}}Ac = A^{\mathrm{T}}y . \tag{31.33}$$

In statistica questa è chiamata ***equazione normale***. La matrice $A^{\mathrm{T}}A$ risulta simmetrica, per l'Esercizio 31.1-3, e se A ha rango di colonna pieno, allora $A^{\mathrm{T}}A$ è anche definita positiva. Quindi, $(A^{\mathrm{T}}A)^{-1}$ esiste e la soluzione all'equazione (31.33) è

$$\begin{aligned} c &= ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})y \\ &= A^{+}y , \end{aligned} \tag{31.34}$$

dove la matrice $A^{+} = ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})$ è chiamata la ***pseudoinversa*** della matrice A. La pseudoinversa è una semplice generalizzazione della nozione di matrice inversa al caso in cui A non sia quadrata. (Si confronti l'equazione (31.34) come soluzione approssimata di $Ac = y$ con la soluzione $A^{-1}b$ come soluzione esatta di $Ax = b$.)

Come esempio di applicazione dei minimi quadrati, si supponga di avere i seguenti cinque punti

$$(-1,2), (1,1), (2,1), (3,0), (5,3) ,$$

mostrati in nero nella figura 31.3. Si desidera trovare un polinomio quadratico

$$F(x) = c_1 + c_2x + c_3x^2$$

che interpoli tali punti.

Si comincia con la matrice dei valori della funzione di base

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix} ,$$

Figura 31.3 *L'approssimazione con i minimi quadrati di un insieme di punti* {(-1,2), (1,1), (2,1), (3,0), (5,3)} *con un polinomio quadratico. I puntini in nero rappresentano i punti dati e i puntini bianchi sono i loro valori approssimati da* $F(x) = 1.2 - 0.757x + 0.214x^2$, *il polinomio quadratico che minimizza la somma degli errori quadratici. L'errore per ciascun punto è mostrato con una linea grigia.*

la cui pseudoinversa è

$$A^{+} = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix} .$$

Moltiplicando y per A^{+}, si ottiene il vettore dei coefficienti

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix} ,$$

che corrisponde al polinomio quadratico

$$F(x) = 1.200 - 0.757x + 0.214x^2$$

che è la migliore approssimazione quadratica dei punti dati rispetto al metodo dei minimi quadrati.

Da un punto di vista pratico, si risolve l'equazione normale (31.33) moltiplicando y per A^{T} e trovando poi una fattorizzazione LU di $A^{\mathrm{T}}A$. Se A ha rango pieno, è garantito che la matrice $A^{\mathrm{T}}A$ sia invertibile perché è simmetrica e definita positiva. (Si veda l'Esercizio 31.1-3 e il Teorema 31.6.)

Esercizi

31.6-1 Si dimostri che ogni elemento della diagonale di una matrice simmetrica definita positiva è positivo.

31.6-2 Sia $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ una matrice simmetrica 2×2 definita positiva. Si dimostri che il suo determinante $ac - b^2$ è positivo per il "completamento del quadrato", in modo simile a quello usato nella dimostrazione del lemma 31.15.

31.6-3 Si dimostri che l'elemento massimo di una matrice simmetrica definita positiva sta sulla diagonale.

31.6-4 Si dimostri che il determinante di ciascuna sottomatrice portante di una matrice simmetrica definita positiva è positivo.

31.6-5 Si denoti con A_k la k-esima sottomatrice portante di una matrice A simmetrica definita positiva. Si dimostri che $\det(A_k)/\det(A_{k-1})$ è il k-esimo perno durante la fattorizzazione LU, dove per convenzione $\det(A_0) = 1$.

31.6-6 Si trovi la funzione della forma

$$F(x) = c_1 + c_2 x + c_3 x^2$$

che sia la migliore approssimazione rispetto ai minimi quadrati per i punti $(1, 1), (2, 1), (3, 3), (4, 8)$.

31.6-7 Si mostri che la pseudoinversa A^+ soddisfa le quattro seguenti equazioni

$$\begin{aligned} AA^+A &= A, \\ A^+AA^+ &= A^+, \\ (AA^+)^T &= AA^+, \\ (A^+A)^T &= A^+A. \end{aligned}$$

Problemi

31-1 L'algoritmo di Shamir per la moltiplicazione di matrici booleane

Nel paragrafo 31.3, si è osservato che l'algoritmo di Strassen per la moltiplicazione di matrici non può essere applicato direttamente alla moltiplicazione di matrici booleane perché il quasianello $Q = (\{0, 1\}, \vee, \wedge, 0, 1)$ non è un anello. Il Teorema 31.10 ha dimostrato che se si usano operazioni aritmetiche su parole di $O(\lg n)$ bit, si potrebbe comunque applicare il metodo di Strassen per moltiplicare matrici booleane $n \times n$ in tempo $O(n^{\lg 7})$. In questo problema si esamina un metodo probabilistico che usa soltanto operazioni sui bit per raggiungere un'efficienza paragonabile ma con qualche piccolo margine di errore.

a. Si mostri che $R = (\{0, 1\}, \oplus, \wedge, 0, 1)$, dove $\oplus$ è la funzione XOR (or esclusivo), è un anello.

Siano $A = (a_{ij})$ e $B = (b_{ij})$ due matrici booleane $n \times n$ e sia $C = (c_{ij}) = AB$ nel quasianello Q. Si generi $A' = (a'_{ij})$ da A usando la seguente procedura randomizzata:

- se $a_{ij} = 0$, allora sia $a'_{ij} = 0$.
- se $a_{ij} = 1$, allora sia $a'_{ij} = 1$ con probabilità $1/2$ e sia $a'_{ij} = 0$ con probabilità $1/2$. Le scelte casuali sono indipendenti per ogni elemento.

b. Sia $C' = (c'_{ij}) = A'B$ nell'anello R. Si mostri che $c_{ij} = 0$ implica $c'_{ij} = 0$. Si mostri che $c_{ij} = 1$ implica $c'_{ij} = 1$ con probabilità $1/2$.

c. Si mostri che, per qualsiasi $\varepsilon > 0$, la probabilità che un dato c'_{ij} non assuma mai i valori c_{ij} per $\lg(n^2/\varepsilon)$ scelte indipendenti della matrice A' è al più ε/n^2. Si mostri che la probabilità che tutte le c'_{ij} assumano i loro valori corretti almeno una volta è almeno $1 - \varepsilon$.

d. Si dia un algoritmo randomizzato di tempo $O(n^{\lg 7} \lg n)$ che calcoli il prodotto di due matrici $n \times n$ nel quasianello booleano Q con probabilità almeno $1 - 1/n^k$ per qualsiasi costante positiva k. Le uniche operazioni consentite sugli elementi delle matrici sono $\vee$, $\wedge$ e $\oplus$.

31-2 Sistemi tridiagonali di equazioni lineari

Si consideri la matrice tridiagonale

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

a. Si trovi una fattorizzazione LU per A.

b. Si risolva l'equazione $Ax = (1\ 1\ 1\ 1\ 1)^T$ usando le sostituzioni in avanti e a ritroso.

c. Si trovi l'inversa di A.

d. Si mostri che per qualsiasi matrice A di dimensione $n \times n$ tridiagonale, simmetrica, definita positiva e per qualsiasi vettore b di dimensione n, l'equazione $Ax = b$ può essere risolta in tempo $O(n)$ eseguendo una fattorizzazione LU. Si deduca che qualunque metodo basato sulla determinazione di A^{-1} nel caso peggiore è asintoticamente più costoso.

e. Si mostri che per qualsiasi matrice di dimensione A $n \times n$ tridiagonale e invertibile e per qualsiasi vettore b di dimensione n, l'equazione $Ax = b$ può essere risolta in tempo $O(n)$ eseguendo una fattorizzazione LUP.

31-3 Spline

Un metodo pratico per interpolare un insieme di punti con una curva è di usare le ***spline cubiche***. Sia dato un insieme $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ di $n + 1$ coppie di valori rappresentanti punti, dove $x_0 < x_1 < \ldots < x_n$. Si desidera interpolare tale insieme di punti con una curva $f(x)$ cubica a tratti (spline). In altre parole, la curva $f(x)$ è fatta di n polinomi cubici $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ per $i = 0, 1, \ldots, n-1$, dove x cade nell'intervallo $x_i \le x \le x_{i+1}$ e il valore della curva

è dato da $f(x) = f_i(x - x_i)$. I punti x_i ai quali i polinomi cubici sono "attaccati" sono chiamati ***nodi***. Per semplicità, si assumerà che $x_i = i$ per $i = 0, 1, \ldots, n$.

Per garantire la continuità di $f(x)$, si richiede che

$$f(x_i) = f_i(0) = y_i ,$$
$$f(x_{i+1}) = f_i(0) = y_{i+1}$$

per $i = 0, 1, \ldots, n-1$. Per garantire che $f(x)$ non sia eccessivamente spigolosa, si richiede la continuità della derivata prima in ogni nodo.

$$f'(x_{i+1}) = f'_i(1) = f'_{i+1}(0)$$

per $i = 0, 1, \ldots, n-1$.

a. Si supponga che per $i = 0, 1, \ldots, n$ siano date non solo le coppie $\{(x_i, y_i)\}$ dei valori dei punti, ma anche la derivata prima $D_i = f'(x_i)$ in ogni nodo. Si esprima ogni coefficiente a_i, b_i, c_i e d_i in termini dei valori y_i, y_{i+1}, D_i e D_{i+1}. (Si ricordi che $x_i = i$.) Quanto velocemente si possono calcolare i $4n$ coefficienti a partire dalle coppie di valori dei punti e dalle derivate prime?

Rimane la questione di come scegliere la derivata prima di $f(x)$ nei nodi. Un metodo è quello di richiedere che la derivata seconda sia continua nei nodi:

$$f''(x_{i+1}) = f''_i(1) = f''_{i+1}(0)$$

per $i = 0, 1, \ldots, n-1$. Per il primo e per l'ultimo nodo, si assume che $f''(x_0) = f''_0(0) = 0$ e $f''(x_n) = f''_n(1) = 0$; queste ipotesi rendono $f(x)$ una splina cubica ***naturale***.

b. Si usino i vincoli di continuità sulla derivata seconda per mostrare che per $i = 1, 2, \ldots, n-1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) . \tag{31.35}$$

c. Si mostri che

$$2D_0 + D_1 = 3(y_1 - y_0) , \tag{31.36}$$
$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}) . \tag{31.37}$$

d. Si riscrivano le equazioni (31.35)-(31.37) come un'equazione matriciale con il vettore $D = \langle D_0, D_1, \ldots, D_n \rangle$ di incognite. Quali sono gli attributi che dovrebbe avere la matrice della nuova equazione?

e. Si deduca che un insieme di $n + 1$ coppie di valori possono essere interpolate con una splina cubica naturale in tempo $O(n)$ (si veda il Problema 31-2).

f. Si mostri come determinare una splina cubica naturale che rappresenti l'interpolazione di un insieme di $n + 1$ punti (x_i, y_i) che soddisfano $x_0 < x_1 < \ldots < x_n$ anche quando x_i non è necessariamente uguale a i. Quale equazione matriciale deve essere risolta e quanto è veloce l'algoritmo progettato?

Note al capitolo

Sono disponibili molti ottimi testi che descrivono il calcolo scientifico e numerico in modo molto più approfondito di quanto sia stato fatto in questo libro. Particolarmente interessanti sono i seguenti: George e Liu [81], Golub e Van Loan [89], Press, Flannery, Teukolsky e Vetterling [161, 162], e Strang [181, 182].

La pubblicazione dell'algoritmo di Strassen nel 1969 [183] suscitò molto interesse. Prima di allora, era difficile immaginare che algoritmi semplici potessero essere migliorati ulteriormente. Il limite asintotico superiore della moltiplicazione tra matrici da allora è stato considerevolmente migliorato. L'algoritmo per moltiplicare matrici $n \times n$ finora asintoticamente più efficiente è dovuto a Coppersmith e Winograd [52] ed ha tempo di esecuzione $O(n^{2.376})$. La rappresentazione grafica dell'algoritmo di Strassen è dovuta a Paterson [155]. Fischer e Meyer [67] adattarono l'algoritmo di Strassen per le matrici booleane (Teorema 31.10).

L'eliminazione di Gauss, sulla quale sono basate le fattorizzazioni LU e LUP, fu il primo metodo sistematico per risolvere sistemi di equazioni lineari. Fu anche uno dei primi algoritmi numerici. Sebbene fosse conosciuto anche precedentemente, la sua scoperta è attribuita a C.F. Gauss (1777-1855). Nella sua famosa pubblicazione [183], Strassen mostrò anche che una matrice $n \times n$ può essere invertita in tempo $O(n^{\lg 7})$. Winograd [203] provò originariamente che la moltiplicazione tra matrici non è più gravosa dell'inversione di matrici, il viceversa è dovuto a Aho, Hopcroft e Ullman [4].

Strang [182] fa una ottima presentazione delle matrici simmetriche definite positive e dell'algebra lineare in generale. Egli fa la seguente osservazione alla pagina 334: "I miei studenti spesso mi chiedono informazioni sulle matrici *asimmetriche* definite positive. Io non uso mai quel termine."

Polinomi e FFT

32

L'addizione di due polinomi di grado n con il metodo semplice richiede tempo $\Theta(n)$, mentre la moltiplicazione con il metodo semplice richiede tempo $\Theta(n^2)$. In questo capitolo si mostrerà come le trasformate veloci di Fourier, o FFT (dall'inglese Fast Fourier Transform), possano ridurre il tempo della moltiplicazione tra polinomi a $\Theta(n \lg n)$.

Polinomi

Un polinomio nella variabile x su un campo algebrico F è una funzione $A(x)$ che può essere rappresentata come segue:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ .$$

Il valore n è detto ***grado-limite*** del polinomio e i valori $a_0, a_1, \ldots, a_{n-1}$ sono detti ***coefficienti*** del polinomio. I coefficienti sono presi dal campo F, tipicamente l'insieme $\mathbb{C}$ dei numeri complessi. Un polinomio $A(x)$ si dice di ***grado*** k se il suo più grande coefficiente diverso da zero è a_k. Il grado di un polinomio di grado-limite n può essere qualsiasi intero compreso tra 0 ed $n - 1$ inclusi. Viceversa, un polinomio di grado k è un polinomio di grado-limite n, per qualsiasi $n > k$.

Varie operazioni possono essere definite sui polinomi. Per quanto riguarda l'***addizione di polinomi***, se $A(x)$ e $B(x)$ sono di grado-limite n, si dice che la loro ***somma*** $C(x)$ è un polinomio, ancora di grado-limite n, tale che $C(x) = A(x) + B(x)$ per tutti i valori di x nel campo. Cioè, se

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

e

$$B(x) = \sum_{j=0}^{n-1} b_j x^j \ ,$$

allora

$$C(x) = \sum_{j=0}^{n-1} c_j x^j \ ,$$

dove $c_j = a_j + b_j$, per $j = 0, 1, \ldots, n-1$. Per esempio, se $A(x) = 6x^3 + 7x^2 - 10x + 9$ e $B(x) = -2x^3 + 4x - 5$, allora $C(x) = 4x^3 + 7x^2 - 6x + 4$.

Per quanto riguarda la ***moltiplicazione di polinomi***, se $A(x)$ e $B(x)$ sono di grado-limite n, il loro ***prodotto*** $C(x)$ è un polinomio di grado-limite $2n-1$ tale che $C(x) = A(x)B(x)$, per tutte le x del campo. Probabilmente il lettore ha già moltiplicato polinomi prima d'ora, moltiplicando ciascun termine di $A(x)$ per ciascun termine di $B(x)$, sommando poi i termini con la stessa potenza. Per esempio, si possono moltiplicare $A(x) = 6x^3 + 7x^2 - 10x + 9$ e $B(x) = -2x^3 + 4x - 5$ come segue

$$
\begin{array}{rrrrrrr}
 & & & 6x^3 & +7x^2 & -10x & +9 \\
 & & & -2x^3 & & +4x & -5 \\
\hline
 & & & -30x^3 & -35x^2 & +50x & -45 \\
 & & 24x^4 & +28x^3 & -40x^2 & +36x & \\
 & -12x^5 & -14x^4 & -18x^3 & & & \\
\hline
 & -12x^6 & -14x^5 & +44x^4 & -20x^3 & -75x^2 & +86x \quad -45
\end{array}
$$

Un altro modo di esprimere il prodotto $C(x)$ è

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \ , \tag{32.1}$$

dove

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \ . \tag{32.2}$$

Si noti che grado(C) = grado(A) + grado(B) e ciò implica

grado-limite(C) = grado-limite(A) + grado-limite(B) − 1

≤ grado-limite(A) + grado-limite(B).

Comunque, si parlerà del grado-limite di C come della somma del grado-limite di A e di B, dato che se un polinomio ha grado-limite k allora ha anche grado-limite $k+1$.

Sommario del capitolo

Il paragrafo 32.1 esamina due modi di rappresentare i polinomi: la rappresentazione a coefficienti e la rappresentazione a punti. I metodi semplici per la moltiplicazione dei polinomi – le equazioni (32.1) e (32.2) – richiedono tempo $\Theta(n^2)$ quando i polinomi sono rappresentati nella forma a coefficienti, ma soltanto tempo $\Theta(n)$ quando sono rappresentati nella forma a punti. Tuttavia si possono moltiplicare polinomi usando la rappresentazione a coefficienti in tempo $\Theta(n \lg n)$ con una conversione tra le due rappresentazioni. Per vedere come un tale metodo funzioni si devono prima studiare le radici complesse dell'unità, al paragrafo 32.2. Si esaminano la FFT e la sua inversa, anch'esse descritte al paragrafo 32.2, usate per effettuare le conversioni. Il paragrafo 32.3 mostra come realizzare in modo efficiente la FFT su modelli di calcolo sia sequenziali che paralleli.

In questo capitolo verranno molto utilizzati i numeri complessi: il simbolo i sarà dunque usato esclusivamente per denotare $\sqrt{-1}$.

32.1 Rappresentazione di polinomi

Le rappresentazioni dei polinomi a coefficienti e a punti sono in qualche modo equivalenti: un polinomio nella forma a punti ha un'unica rappresentazione corrispondente nella forma a coefficienti. In questo paragrafo si analizzano le due rappresentazioni e si mostra come possano essere combinate per consentire la moltiplicazione di due polinomi di grado-limite n in tempo $\Theta(n \lg n)$.

Rappresentazione a coefficienti

Una ***rappresentazione a coefficienti*** di un polinomio $A(x) = \sum_{j=0}^{n-1} a_j x^j$ di grado-limite n è un vettore di coefficienti $a = (a_0, a_1, \ldots, a_{n-1})$. In questo capitolo, nelle equazioni matriciali, si tratteranno i vettori come vettori colonna.

La rappresentazione a coefficienti è conveniente per certe operazioni sui polinomi. Per esempio, l'operazione di ***valutazione del polinomio*** $A(x)$ in un dato punto x_0 consiste nel calcolo del valore $A(x_0)$. Usando la ***regola di Horner***, la valutazione impiega tempo $\Theta(n)$:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots + x_0(a_{n-2} + x_0(a_{n-1}))\ldots)) \ .$$

Analogamente, la somma di due polinomi rappresentati dai vettori dei coefficienti $a = (a_0, a_1, \ldots, a_{n-1})$ e $b = (b_0, b_1, \ldots, b_{n-1})$ impiega tempo $\Theta(n)$: si deve calcolare soltanto il vettore dei coefficienti $c = (c_0, c_1, \ldots, c_{n-1})$, dove $c_j = a_j + b_j$ per $j = 0, 1, \ldots, n-1$.

Si consideri ora la moltiplicazione di due polinomi $A(x)$ e $B(x)$ di grado-limite n rappresentati nella forma a coefficienti. Se si usa il metodo descritto dalle equazioni (32.1) e (32.2), la moltiplicazione di polinomi impiega $\Theta(n^2)$ tempo poiché ciascun coefficiente nel vettore a deve essere moltiplicato per ogni coefficiente nel vettore b. L'operazione di moltiplicazione di polinomi nella forma a coefficienti sembra essere considerevolmente più complessa rispetto alla valutazione di un polinomio o alla somma di due polinomi. Il vettore c dei coefficienti risultante, dato dall'equazione (32.2), è chiamato anche la ***convoluzione*** dei vettori a e b di input ed è denotata con $c = a \otimes b$. Poiché la moltiplicazione di polinomi e il calcolo delle convoluzioni sono problemi computazionali fondamentali di considerevole importanza pratica, questo capitolo è incentrato su algoritmi efficienti che risolvono tali problemi.

Rappresentazione a punti

Una ***rappresentazione a punti*** di un polinomio $A(x)$ di grado-limite n è un insieme di ***n coppie punto/valore***

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

tali che tutti gli x_i sono distinti e

$$y_k = A(x_k), \tag{32.3}$$

per $k = 0, 1, \ldots, n-1$. Un polinomio ha molte rappresentazioni a punti diverse poiché qualunque insieme di n punti distinti $x_0, x_1, \ldots, x_{n-1}$ può essere usato come base per la rappresentazione.

Il calcolo di una rappresentazione a punti di un polinomio dato nella forma a coefficienti, in linea di principio, è semplice poiché tutto ciò che si deve fare è selezionare n punti distinti $x_0, x_1, \ldots, x_{n-1}$ e quindi valutare $A(x_k)$ per $k = 0, 1, \ldots, n-1$. Con il metodo di Horner questa valutazione su n punti richiede tempo $\Theta(n^2)$. Si vedrà in seguito che scegliendo intelligentemente gli x_k, questo calcolo può essere reso più veloce fino a un tempo $\Theta(n \lg n)$.

L'inversa della valutazione – la determinazione della forma a coefficienti di un polinomio partendo dalla rappresentazione a punti – è chiamata ***interpolazione***. Il seguente teorema mostra che l'interpolazione è ben definita, assumendo che il grado-limite del polinomio di interpolazione sia uguale al numero di coppie punto/valore date.

Teorema 32.1 (Unicità del polinomio di interpolazione)

Per qualsiasi insieme $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$, esiste un unico polinomio $A(x)$ di grado-limite n tale che $y_k = A(x_k)$ per $k = 0, 1, \ldots, n-1$.

Dimostrazione. La dimostrazione è basata sull'esistenza dell'inversa di una determinata matrice. L'equazione 32.3 è equivalente alla equazione matriciale

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}. \tag{32.4}$$

La matrice sulla sinistra è denotata con $V(x_0, x_1, \ldots, x_{n-1})$ ed è conosciuta come matrice di Vandermonde. Dall'Esercizio 31.1-10, questa matrice ha determinante

$$\prod_{0 \le j < k \le n-1} (x_k - x_j),$$

e perciò, per il Teorema 31.5, essa è invertibile (cioè non singolare) se gli x_k sono distinti. Quindi, i coefficienti a_j possono essere determinati univocamente data la rappresentazione a punti:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y$$

■

La dimostrazione del Teorema 32.1 descrive un algoritmo di interpolazione basato sulla risoluzione dell'insieme di equazioni lineari (32.4). Usando gli algoritmi del Capitolo 31 per la decomposizione LU, queste equazioni possono essere risolte in tempo $O(n^3)$.

Un algoritmo più efficiente per l'interpolazione di n punti è basato sulla ***formula di Lagrange***:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \ne k} (x - x_j)}{\prod_{j=0}^{n-1} (x_k - x_j)}. \tag{32.5}$$

Si può verificare che la parte destra dell'equazione (32.5) è un polinomio di grado-limite n che soddisfa $A(x_k) = y_k$ per tutti i k. L'Esercizio 32.1-4 richiede di trovare un algoritmo per calcolare in tempo $\Theta(n^2)$ i coefficienti di A usando la formula di Lagrange.

Quindi, la valutazione e l'interpolazione di n punti sono operazioni ben definite: una è l'inversa dell'altra e servono per convertire la rappresentazione a coefficienti di un polinomio in una rappresentazione a punti e viceversa[1]. Gli algoritmi per questi problemi impiegano tempo $\Theta(n^2)$.

La rappresentazione a punti è abbastanza conveniente per molte operazioni sui polinomi. Per quanto riguarda l'addizione, se $C(x) = A(x) + B(x)$, allora $C(x_k) = A(x_k) + B(x_k)$ per ogni punto x_k. Più precisamente se si ha una rappresentazione a punti per A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

e per B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

(si noti che A e B sono valutati negli *stessi* n punti), allora una rappresentazione a punti per C è

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\}.$$

Il tempo per addizionare due polinomi di grado n nella forma a punti è pertanto $\Theta(n)$.

Analogamente la rappresentazione a punti è conveniente per la moltiplicazione di polinomi. Se $C(x) = A(x)B(x)$, allora $C(x_k) = A(x_k)B(x_k)$ per ogni punto x_k e si può moltiplicare punto a punto una rappresentazione a punti di A per una rappresentazione a punti di B, ottenendo una rappresentazione a punti di C. Tuttavia si deve considerare il problema che il grado-limite di C è la somma dei gradi-limite di A e B.

Una tipica rappresentazione a punti di A e B è costituita da n coppie punto/valore per ciascun polinomio. Moltiplicando si ottengono n coppie punto/valore per C, ma poiché il grado limite di C è $2n$, il Teorema 32.1 implica la necessità di $2n$ coppie punto/valore per una rappresentazione a punti di C. Si deve allora cominciare con una rappresentazione a punti "estesa" di A e B ognuna consistente di $2n$ coppie punto/valore. Data una rappresentazione a punti estesa di A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\},$$

e una corrispondente rappresentazione a punti estesa di B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\},$$

allora una rappresentazione a punti di C è

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}.$$

Dati due polinomi di input in forma a punti estesa, si ha allora che il tempo per moltiplicarli per ottenere la forma a punti del risultato è $\Theta(n)$, molto meno del tempo richiesto per moltiplicare polinomi nella forma a coefficienti.

[1] L'interpolazione è un problema notoriamente "traditore" dal punto di vista della stabilità numerica. Sebbene gli approcci descritti in questo libro siano corretti matematicamente, piccole differenze nell'input o errori di arrotondamento durante il calcolo possono causare notevoli differenze nei risultati.

Infine si consideri il problema di valutare un polinomio dato nella forma a punti in un nuovo punto. Per questo problema, apparentemente, non vi è approccio più semplice che eseguire prima la conversione del polinomio nella forma a coefficienti e quindi la sua valutazione nel nuovo punto.

Moltiplicazione veloce di polinomi nella forma a coefficienti

Si può usare il metodo di moltiplicazione di polinomi nella forma a punti, che richiede tempo lineare, per rendere più efficiente la moltiplicazione di polinomi nella forma a coefficienti? La risposta sta nella capacità di convertire rapidamente un polinomio dalla forma a coefficienti alla forma a punti (valutazione) e viceversa (interpolazione).

Si può usare qualunque punto si voglia come punto di valutazione ma, con una scelta accurata dei punti di valutazione, si può passare da una rappresentazione all'altra in tempo $O(n \lg n)$. Come si vedrà nel paragrafo 32.2, se si scelgono "radici complesse dell'unità" come punti di valutazione, si può produrre una rappresentazione a punti prendendo la trasformata discreta di Fourier (o DFT dall'inglese Discrete Fourier Transform) di un vettore di coefficienti. L'operazione inversa, l'interpolazione, può essere eseguita prendendo la "DFT inversa" delle coppie punto/valore, ottenendo un vettore di coefficienti. Il paragrafo 32.2 mostrerà come la FFT esegua le operazioni DFT e DFT inversa in tempo $\Theta(n \lg n)$.

La figura 32.1 mostra graficamente questa strategia. Ancora un piccolo dettaglio riguarda il grado-limite. Il prodotto di due polinomi di grado-limite n è un polinomio di grado-limite $2n$. Prima di valutare i polinomi in input A e B, allora, si raddoppia il loro grado-limite a $2n$ aggiungendo n coefficienti 0 di ordine più alto. Poiché i vettori hanno $2n$ elementi, si usano le "$(2n)$-esime radici complesse dell'unità", denotate nella figura 32.1 con i termini ω_{2n}.

Data la FFT, si ha la seguente procedura di tempo $\Theta(n \lg n)$ per moltiplicare due polinomi $A(x)$ e $B(x)$ di grado-limite n, dove le rappresentazioni dell'input e dell'output sono nella forma a coefficienti. Si assume che n sia una potenza di 2; questa richiesta può sempre essere soddisfatta aggiungendo coefficienti 0 di ordine più alto.

1. *Doppio grado-limite:* Si creano le rappresentazioni a coefficienti di $A(x)$ e $B(x)$ come polinomi di grado-limite $2n$ aggiungendo ad ognuno n coefficienti 0 di grado più alto.
2. *Valutazione:* Si calcolano le rappresentazioni a punti di $A(x)$ e $B(x)$ di lunghezza $2n$ attraverso due applicazioni di una FFT di ordine $2n$. Queste rappresentazioni contengono i valori dei due polinomi nelle radici $(2n)$-esime dell'unità.
3. *Moltiplicazione punto a punto:* Si calcola una rappresentazione a punti del polinomio $C(x) = A(x)\,B(x)$ moltiplicando questi valori punto a punto. Questa rappresentazione contiene il valore di $C(x)$ in ognuna delle radici $(2n)$-esime dell'unità.
4. *Interpolazione:* Si crea la rappresentazione a coefficienti del polinomio $C(x)$ attraverso una singola applicazione di una FFT su $2n$ coppie punto/valore per calcolare la DFT inversa.

I passi (1) e (3) richiedono tempo $\Theta(n)$ e i passi (2) e (4) richiedono tempo $\Theta(n \lg n)$. Pertanto, dopo aver mostrato come usare la FFT, avremo dimostrato il seguente teorema.

Figura 32.1 *Uno schema del metodo efficiente di moltiplicazione di polinomi. Le rappresentazioni in alto sono nella forma a coefficienti, mentre quelle in basso sono nella forma a punti. Le frecce da sinistra a destra corrispondono all'operazione di moltiplicazione. I termini ω_{2n} sono le $(2n)$-esime radici complesse dell'unità.*

Teorema 32.2

Il prodotto di due polinomi di grado-limite n può essere calcolato in tempo $\Theta(n \lg n)$. ■

Esercizi

32.1-1 Si moltiplichino i polinomi $A(x) = 7x^3 - x^2 + x - 10$ e $B(x) = 8x^3 - 6x + 3$ usando le equazioni (32.1) e (32.2).

32.1-2 La valutazione di un polinomio $A(x)$ di grado-limite n in un dato punto x_0 può essere fatta anche dividendo $A(x)$ per il polinomio $(x - x_0)$ per ottenere un polinomio quoziente $q(x)$ di grado-limite $n - 1$ e un resto r, tale che

$$A(x) = q(x)(x - x_0) + r\,.$$

Chiaramente, $A(x_0) = r$. Si mostri come calcolare il resto r e i coefficienti di $q(x)$ in tempo $\Theta(n)$ avendo x_0 e i coefficienti di A.

32.1-3 Si derivi una rappresentazione a punti per $A^{rev}(x) = \sum_{j=0}^{n-1} a_{n-1-j}x^j$ a partire da una rappresentazione a punti per $A(x) = \sum_{j=0}^{n-1} a_j x^j$, assumendo che nessuno dei punti sia 0.

32.1-4 Si mostri come usare l'equazione (32.5) per interpolare in tempo $\Theta(n^2)$. (*Suggerimento:* prima si calcolino $\prod_j (x - x_k)$ e $\prod_j (x_j - x_k)$ e quindi si divida ogni termine per $(x - x_k)$ e $(x_j - x_k)$ in maniera opportuna. Si veda l'Esercizio 32.1-2.)

32.1-5 Spiegare cosa è sbagliato nell'"ovvio" approccio alla divisione polinomiale quando si usa una rappresentazione a punti. Si discuta separatamente il caso in cui la divisione riesce in modo esatto e il caso in cui ciò non avviene.

32.1-6 Si considerino due insiemi A e B ognuno avente n interi nell'intervallo da 0 a $10n$. Si desidera calcolare la ***somma Cartesiana*** di A e B definita da

$$C = \{x + y : x \in A \text{ e } y \in B\} .$$

Si noti che gli interi in C sono nell'intervallo da 0 a $20n$. Si vogliono trovare gli elementi di C e il numero di volte che ogni elemento di C è realizzato come somma di elementi in A e B. Si mostri che il problema può essere risolto in tempo $O(n \lg n)$. (*Suggerimento*: si rappresentino A e B come polinomi di grado $10n$.)

32.2 DFT e FFT

Nel paragrafo 32.1 si è detto che se si usano radici complesse dell'unità si può valutare e interpolare in tempo $\Theta(n \lg n)$. In questo paragrafo, si definiscono le radici complesse dell'unità e se ne studiano le proprietà, si definisce la DFT e poi si mostra come la FFT calcola la DFT e la sua inversa in tempo $\Theta(n \lg n)$.

Radici complesse dell'unità

La ***radice n-esima complessa dell'unità*** è un numero complesso ω tale che

$$\omega^n = 1 .$$

Vi sono esattamente n radici n-esime complesse dell'unità; queste sono $e^{2\pi ik/n}$ per $k = 0, 1, \ldots, n-1$. Per interpretare questa formula, si usa la definizione dell'esponenziale di un numero complesso:

$$e^{iu} = \cos(u) + i \operatorname{sen}(u) .$$

La figura 32.2 mostra che le n radici complesse dell'unità sono disposte in modo equidistante sul cerchio di raggio unitario centrato nell'origine del piano complesso. Il valore

$$\omega_n = e^{2\pi i/n} \tag{32.6}$$

è chiamato la ***radice primitiva n-esima dell'unità***: tutte le altre radici n-esime dell'unità sono potenze di ω_n.

Le n radici n-esime complesse dell'unità,

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1} ,$$

formano un gruppo rispetto alla moltiplicazione (si veda il paragrafo 33.3). Questo gruppo ha la stessa struttura del gruppo additivo $(\mathbb{Z}_n, +)$ modulo n, poiché $\omega_n^n = \omega_n^0 = 1$ implica $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Analogamente, $\omega_n^{-1} = \omega_n^{n-1}$. Le proprietà essenziali delle radici n-esime complesse dell'unità sono date nei seguenti lemmi.

Lemma 32.3 (Lemma di cancellazione)
Per qualsiasi intero $n \geq 0$, $k \geq 0$ e $d \geq 0$

$$\omega_{dn}^{dk} = \omega_n^k . \tag{32.7}$$

Figura 32.2 *I valori di $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ nel piano complesso dove $\omega_n = e^{2\pi i/8}$ è la radice primitiva ottava dell'unità.*

Dimostrazione. Il lemma segue direttamente dall'equazione (32.6) dato che

$$\begin{aligned} \omega_{dn}^{dk} &= (e^{2\pi i/dn})^{dk} \\ &= (e^{2\pi i/n})^k \\ &= \omega_n^k . \end{aligned}$$

■

Corollario 32.4
Per qualsiasi intero n pari e maggiore di 0, vale

$$\omega_n^{n/2} = \omega_2 = -1 .$$

Dimostrazione. La dimostrazione è lasciata come Esercizio 32.2-1. ■

Lemma 32.5 (Lemma di dimezzamento)
Se $n > 0$ è pari, allora il quadrato delle n radici n-esime complesse dell'unità sono le $n/2$ radici $n/2$-esime complesse dell'unità.

Dimostrazione. Per il lemma di cancellazione, si ha $(\omega_n^k)^2 = \omega_{n/2}^k$, per qualunque intero k non negativo. Si noti che se si calcola il quadrato di tutte le radici n-esime complesse dell'unità, allora ciascuna radice $n/2$-esima dell'unità è ottenuta esattamente due volte, in quanto

$$\begin{aligned} (\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k} \omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2 . \end{aligned}$$

Pertanto, ω_n^k e $\omega_n^{k+n/2}$ hanno lo stesso quadrato. Questa proprietà può essere dimostrata anche

usando il corollario 32.4, dato che $\omega_n^{n/2} = -1$ implica $\omega_n^{k+n/2} = -\omega_n^k$ e pertanto $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$. ■

Come si vedrà, il lemma di dimezzamento è essenziale all'approccio divide-et-impera usato per eseguire le conversioni tra le rappresentazioni di polinomi a coefficienti e a punti: infatti, tale lemma garantisce che i sottoproblemi ricorsivi siano grandi al più la metà del problema originario.

Lemma 32.6 (Lemma della sommatoria)

Per qualsiasi intero $n \geq 1$ e intero k non negativo e non divisibile per n,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0 .$$

Dimostrazione. Dato che l'equazione (3.3) si applica ai valori complessi,

$$\begin{aligned}\sum_{j=0}^{n-1} (\omega_n^k)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\ &= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\ &= \frac{(1)^k - 1}{\omega_n^k - 1} \\ &= 0 .\end{aligned}$$

Richiedendo che k non sia divisibile per n si ha che il denominatore è diverso da 0 dato che $\omega_n^k = 1$ soltanto quando k è divisibile per n. ■

DFT

Si ricorda che si vuol valutare un polinomio

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

di grado-limite n in $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ (cioè nelle n radici n-esime complesse dell'unità) [2]. Senza perdita di generalità, si assume che n sia una potenza di 2 poiché un dato grado-limite può sempre essere aumentato – si possono sempre aggiungere nuovi coefficienti 0 di grado più alto. Si assume che A sia dato nella forma a coefficienti: $a = (a_0, a_1, \ldots, a_{n-1})$.

Si definiscono i risultati y_k, per $k = 0, 1, \ldots, n-1$, come

[2] Nel paragrafo 32.1 ci si è riferiti alla lunghezza n in realtà come $2n$ poiché si raddoppia il grado-limite di un dato polinomio prima della valutazione. Perciò, nel contesto della moltiplicazione di polinomi in realtà si opera con radici $(2n)$-esime complesse dell'unità.

$$\begin{aligned}y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj} .\end{aligned} \tag{32.8}$$

Il vettore $y = (y_0, y_1, \ldots, y_{n-1})$ è la ***trasformata discreta di Fourier (DFT)*** del vettore dei coefficienti $a = (a_0, a_1, \ldots, a_{n-1})$. Si scrive anche $y = \mathrm{DFT}_n(a)$.

FFT

Usando un metodo conosciuto come ***trasformata veloce di Fourier (FFT)***, che trae vantaggio dalle speciali proprietà delle radici complesse dell'unità, si può calcolare $\mathrm{DFT}_n(a)$ in tempo $\Theta(n \lg n)$ in contrapposizione al tempo $\Theta(n^2)$ del metodo banale.

Il metodo FFT impiega una strategia divide-et-impera usando i coefficienti di $A(x)$ di indice pari e di indice dispari separatamente per definire due nuovi polinomi $A^{[0]}(x)$ e $A^{[1]}(x)$ di grado-limite $n/2$:

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1} .$$
$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1} .$$

Si noti che $A^{[0]}$ contiene tutti i coefficienti di A di indice pari (La rappresentazione binaria dell'indice termina con il bit 0) e $A^{[1]}$ contiene tutti i coefficienti di A di indice dispari (La rappresentazione binaria dell'indice termina con il bit 1).

Ne segue che

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) , \tag{32.9}$$

così che il problema della valutazione di $A(x)$ in $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ si riduce a

1. valutare i polinomi $A^{[0]}(x)$ e $A^{[1]}(x)$ di grado-limite $n/2$ nei punti

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 , \tag{32.10}$$

   e poi
2. combinare i risultati in accordo all'equazione (32.9).

Dal lemma di dimezzamento, la lista di valori (32.10) consiste non di n valori distinti ma solo delle $n/2$ radici $(n/2)$-esime complesse dell'unità, dove ogni radice si ripete esattamente due volte. Allora i polinomi $A^{[0]}(x)$ e $A^{[1]}(x)$ di grado-limite $n/2$ sono valutati ricorsivamente nelle $n/2$ radici $(n/2)$-esime complesse dell'unità. Questi sottoproblemi hanno esattamente la stessa forma del problema originale ma la loro dimensione è la metà. Si è diviso con successo il calcolo di un DFT_n di n elementi nel calcolo di due $\mathrm{DFT}_{n/2}$ di $n/2$ elementi. Questa decomposizione è la base per il seguente algoritmo ricorsivo FFT che calcola la DFT del vettore $a = (a_0, a_1, \ldots, a_{n-1})$ di n elementi, con n potenza di 2.

```
RECURSIVE-FFT(A)
n ← length[a]          ▷ n è una potenza di 2.
if n = 1
        then return a
```

4 $\omega_n \leftarrow e^{2\pi i/n}$
5 $\omega \leftarrow 1$
6 $a^{[0]} = (a_0, a_2, \ldots, a_{n-2})$
7 $a^{[1]} = (a_1, a_3, \ldots, a_{n-1})$
8 $y^{[0]} \leftarrow$ RECURSIVE-FFT$(a^{[0]})$
9 $y^{[1]} \leftarrow$ RECURSIVE-FFT$(a^{[1]})$
10 **for** $k \leftarrow 0$ **to** $n/2 - 1$
11 **do** $y_k \leftarrow y_k^{[0]} + \omega y_k^{[1]}$
12 $y_{k+(n/2)} \leftarrow y_k^{[0]} - \omega y_k^{[1]}$
13 $\omega \leftarrow \omega\omega_n$
14 **return** y ▷ si suppone che y sia un vettore colonna.

La procedura RECURSIVE-FFT funziona come segue. Le linee 2-3 rappresentano la base della ricorsione; la DFT di un elemento è l'elemento stesso, dato che in questo caso

$$\begin{aligned} y_0 &= a_0\omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 . \end{aligned}$$

Le linee 6-7 definiscono il vettore dei coefficienti per i polinomi $A^{[0]}$ e $A^{[1]}$. Le linee 4, 5 e 13 garantiscono che ω sia aggiornato in modo appropriato così che quando le linee 11-12 vengono eseguite si ha $\omega = \omega_n^k$. (Mantenere il valore corrente di ω da un'iterazione all'altra fa risparmiare il tempo sul calcolo di ω_n^k che deve solo essere aggiornato ogni volta nel ciclo **for**). Le linee 8-9 eseguono il calcolo ricorsivo $\mathrm{DFT}_{n/2}$, assegnando, per $k = 0, 1, \ldots, n/2 - 1$,

$$y_k^{[0]} = A^{[0]}(\omega_{n/2}^k) ,$$
$$y_k^{[1]} = A^{[1]}(\omega_{n/2}^k) ,$$

oppure, poiché $\omega_{n/2}^k = \omega_n^{2k}$ per il lemma di cancellazione,

$$y_k^{[0]} = A^{[0]}(\omega_n^{2k}) ,$$
$$y_k^{[1]} = A^{[1]}(\omega_n^{2k}) .$$

Le linee 11-12 combinano i risultati del calcolo ricorsivo di $\mathrm{DFT}_{n/2}$. Per $y_0, y_1, \ldots, y_{n/2-1}$, dalla linea 11 si ha

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) , \end{aligned}$$

dove l'ultima linea segue dall'equazione (32.9). Per $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$ e per $k = 0, 1, \ldots, n/2 - 1$, dalla linea 12 si ha

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) \\ &= A(\omega_n^{k+(n/2)}) . \end{aligned}$$

La seconda linea segue dalla prima poiché $\omega_n^{k+(n/2)} = -\omega_n^k$. La quarta linea segue dalla terza perché $\omega_n^n = 1$ implica $\omega_n^{2k} = \omega_n^{2k+n}$. L'ultima linea segue dall'equazione (32.9). Pertanto, il vettore y restituito da RECURSIVE-FFT è in effetti la DFT del vettore di input a.

Per determinare il tempo di esecuzione della procedura RECURSIVE-FFT, si noti che, senza considerare le chiamate ricorsive, ogni invocazione richiede $\Theta(n)$, dove n è la lunghezza del vettore di input. La ricorrenza per il tempo di esecuzione è allora

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n). \end{aligned}$$

Pertanto, si può valutare un polinomio di grado-limite n nelle radici n-esime complesse dell'unità in tempo $\Theta(n \lg n)$ usando la trasformata veloce di Fourier.

Interpolazione nelle radici complesse dell'unità

Si completa ora lo schema di moltiplicazione polinomiale mostrando come interpolare le radici complesse dell'unità con un polinomio, il che consente di convertire una forma a punti in una forma a coefficienti. Si interpola scrivendo la DFT come una equazione matriciale e poi considerando la forma della matrice inversa.

Dall'equazione (32.4), si può scrivere la DFT come la matrice prodotto $y = V_n a$, dove V_n è una matrice di Vandermonde contenente le potenze appropriate di ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \ldots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \ldots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \ldots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ldots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \ldots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

L'elemento (k, j) di V_n è ω_n^{kj}, per $j, k = 0, 1, \ldots, n-1$, e gli esponenti degli elementi di V_n formano una tabella di moltiplicazione.

Per l'operazione inversa, che sarà indicata con $a = \mathrm{DFT}_n^{-1}(y)$, si procede moltiplicando y per la matrice V_n^{-1}, l'inversa di V_n.

Teorema 32.7

Per $j, k = 0, 1, \ldots, n-1$, l'elemento (j, k) di V_n^{-1} è ω_n^{-kj}/n.

Dimostrazione. Si mostra che $V_n^{-1}V_n = I_n$, la matrice identità $n \times n$. Si consideri l'elemento (j, j') di $V_n^{-1}V_n$:

$$[V_n^{-1}V_n]_{jj'} = \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'})$$
$$= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n .$$

Questa sommatoria è uguale a 1 se $j' = j$, mentre vale 0 altrimenti, per il lemma della sommatoria (Lemma 32.6). Si noti che, affinché sia applicabile il lemma della sommatoria, si conta sul fatto che $-(n-1) < j'-j < n-1$, così che $j'-j$ non sia divisibile per n. ■

Data la matrice inversa V_n^{-1}, si ha che $\mathrm{DFT}_n^{-1}(y)$ è data da

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{32.11}$$

per $j = 1, 2, \ldots, n-1$. Confrontando le equazioni (32.8) e (32.11) si vede che, se si modifica l'algoritmo FFT per scambiare i ruoli di a e y, sostituendo ω con ω_n^{-1} e dividendo ogni elemento del risultato per n, si calcola la DFT inversa (si veda l'Esercizio 32.2-4). Pertanto anche DFT_n^{-1} può essere calcolata in tempo $\Theta(n \lg n)$.

Pertanto, usando la FFT e la FFT inversa, si può trasformare la rappresentazione a punti di un polinomio di grado-limite n in quella a coefficienti, o viceversa, in tempo $\Theta(n \lg n)$. Nel contesto della moltiplicazione di polinomi, si è mostrato il seguente teorema.

Teorema 32.8 (Teorema di convoluzione)

Dati due vettori a e b qualsiasi di lunghezza n, dove n è una potenza di 2.

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b)) ,$$

dove i vettori a e b sono estesi con degli 0 fino alla lunghezza $2n$ e $\cdot$ denota il prodotto componente per componente di due vettori di $2n$ elementi. ■

Esercizi

32.2-1 Si dimostri il corollario 32.4.

32.2-2 Si calcoli la DFT del vettore (0, 1, 2, 3).

32.2-3 Si ripeta l'Esercizio 32.1-1 usando il metodo di tempo $\Theta(n \lg n)$.

32.2-4 Si scriva lo pseudocodice per calcolare DFT_n^{-1} in tempo $\Theta(n \lg n)$.

32.2-5 Si descriva la generalizzazione della procedura FFT al caso in cui n sia una potenza di 3. Si dia una ricorrenza per il tempo di esecuzione e si risolva la ricorrenza.

* *32.2-6* Si supponga che invece di eseguire una FFT di n elementi (dove n è pari) sul campo dei numeri complessi, si usi l'anello $\mathbf{Z}_m$ di interi modulo m, dove $m = 2^{tn/2} + 1$ e t è un intero positivo arbitrario. Si usi $w = 2^t$ invece di ω_n come radice primitiva n-esima dell'unità modulo m. Si dimostri che la DFT e la DFT inversa sono ben definite in questo sistema.

32.2-7 Data una lista di valori $z_0, z_1, \ldots, z_{n-1}$ (eventualmente con ripetizioni), si mostri come trovare i coefficienti del polinomio $P(x)$ di grado-limite n che ha 0 solo in $z_0, z_1, \ldots, z_{n-1}$ (eventualmente con ripetizioni). La procedura dovrebbe impiegare tempo $O(n \lg^2 n)$. (*Suggerimento:* il polinomio $P(x)$ ha uno zero in z_j se e solo se $P(x)$ è un multiplo di $(x - z_j)$.)

* *32.2-8* La ***trasformata rilassata*** di un vettore $a = (a_0, a_1, \ldots, a_{n-1})$ è il vettore $y = (y_0, y_1, \ldots, y_{n-1})$ dove $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ e z è un numero complesso qualsiasi. La DFT è allora un caso speciale della trasformata rilassata, ottenuta prendendo $z = \omega_n$. Si dimostri che la trasformata rilassata può essere valutata in tempo $O(n \lg n)$ per qualunque numero complesso z. (*Suggerimento:* si usi l'equazione

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right)\left(z^{-(k-j)^2/2}\right)$$

per vedere la trasformata rilassata come una convoluzione.)

32.3 Realizzazioni efficienti della FFT

Dato che le applicazioni pratiche della DFT, come ad esempio l'elaborazione di segnali, richiedono la massima velocità, questo paragrafo esaminerà due realizzazioni efficienti per la FFT. Si esamina dapprima una versione iterativa dell'algoritmo FFT che impiega tempo $\Theta(n \lg n)$ ma che ha una costante nascosta nella notazione Θ più piccola di quella della realizzazione ricorsiva del paragrafo 32.2. Quindi, si useranno i concetti intuitivi che portano alla realizzazione iterativa per progettare un circuito parallelo efficiente per la FFT.

Realizzazione iterativa della FFT

Si noti innanzitutto che il ciclo **for** alle linee 10-13 di RECURSIVE-FFT prevede il calcolo di $\omega_n^k y_k^{[1]}$ due volte. Nella terminologia dei compilatori, questo valore è conosciuto come ***sottoespressione comune***. Si può modificare il ciclo per calcolarlo una sola volta, memorizzandolo in una variabile temporanea t.

```
for k ← 0 to n/2 − 1
    do t ← ω y_k^[1]
       y_k ← y_k^[0] + t
       y_{k+(n/2)} ← y_k^[0] − t
       ω ← ωω_n
```

Figura 32.3 *Un'operazione a farfalla. I due valori di input entrano da sinistra, ω_n^k è moltiplicato per $y_k^{[1]}$, e la somma e la differenza sono restituite come output a destra. La figura può essere interpretata come un circuito combinatorio.*

Figura 32.4 *L'albero dei vettori di input alla chiamata ricorsiva della procedura* RECURSIVE-FFT. *La chiamata iniziale è per $n = 8$.*

L'insieme di operazioni di questo ciclo – la moltiplicazione di ω (che è uguale a ω_n^k) per $y_k^{[1]}$, l'assegnamento del prodotto a t, la somma e la sottrazione di t da $y_k^{[0]}$ – è conosciuto come ***operazione a farfalla*** ed è mostrato schematicamente nella figura 32.3.

Mostriamo ora come rendere la struttura dell'algoritmo FFT iterativa piuttosto che ricorsiva. Nella figura 32.4, i vettori di input alle chiamate ricorsive in una invocazione di RECURSIVE-FFT sono stati sistemati in una struttura ad albero dove la chiamata iniziale è per $n = 8$. L'albero ha un nodo per ogni chiamata della procedura, etichettato con il corrispondente vettore di input. Ogni invocazione ricorsiva di RECURSIVE-FFT esegue due chiamate ricorsive a meno che non abbia ricevuto un vettore di un solo elemento. Si esegue la prima chiamata al figlio sinistro e la seconda al figlio destro.

Osservando l'albero si può notare che, se si potessero sistemare gli elementi del vettore iniziale a nell'ordine in cui essi appaiono sulle foglie, si potrebbe simulare l'esecuzione della procedura RECURSIVE-FFT come segue. Prima si prendono gli elementi in coppie, si calcola la DFT di ogni coppia usando un'operazione a farfalla e si sostituisce la coppia con la sua DFT. Quindi il vettore conterrà $n/2$ DFT di 2 elementi. Poi si prendono questi $n/2$ DFT a coppie e si calcola la DFT dei quattro elementi originati dall'esecuzione di due operazioni a farfalla, sostituendo le due DFT di 2 elementi con una DFT di 4 elementi. Il vettore quindi contiene $n/4$ DFT di 4 elementi. Si continua così finché il vettore conterrà due DFT di $n/2$ elementi che possono essere combinati mediante $n/2$ operazioni a farfalla per ottenere la DFT degli n elementi.

Per tradurre quest'osservazione nel codice, si usa un array $A[0 \,..\, n-1]$ che inizialmente contiene gli elementi del vettore di input a nell'ordine in cui essi compaiono nelle foglie dell'albero di figura 32.4. (Si mostrerà in seguito come determinare quest'ordine.) Poiché la combinazione deve essere fatta ad ogni livello dell'albero, si introduce una variabile s per

contare i livelli che vanno da 1 (in alto, quando si combinano le coppie per formare le DFT di 2 elementi) a $\lg n$ (in basso, quando si combinano due DFT di $n/2$ elementi per produrre il risultato finale). L'algoritmo ha dunque la seguente struttura:

```
1  for s ← 1 to lg n
2      do for k ← 0 to n − 1 by 2^s
3          do combina le due DFT di 2^(s−1) elementi che sono in
               A[k . . k + 2^(s−1) − 1] e A[k + 2^(s−1) . . k + 2^s − 1]
               in una DFT di 2^s elementi che va in A[k . . k + 2^s − 1]
```

Si può esprimere il corpo del ciclo (linea 3) in modo più preciso. Si copia il ciclo for dalla procedura ricorsiva RECURSIVE-FFT, identificando $y^{[0]}$ con $A[k \,..\, k + 2^{s-1} - 1]$ e $y^{[1]}$ con $A[k + 2^{s-1} \,..\, k + 2^s - 1]$. Il valore di ω usato in ogni operazione a farfalla dipende dal valore di s; si usa ω_m, dove $m = 2^s$. (Si introduce la variabile m solo per una maggior leggibilità.) Si introduce un'altra variabile temporanea u che permette di eseguire l'operazione a farfalla in loco. Quando si sostituisce la linea 3 nella struttura complessiva con il corpo del ciclo, si ottiene il seguente pseudocodice che forma la base del nostro algoritmo FFT iterativo finale e della realizzazione parallela che presenteremo in seguito.

```
FFT-BASE(a)
1  n ← length[a]              ▷ n è una potenza di 2.
2  for s ← 1 to lg n
3      do m ← 2^s
4          ω_m ← e^(2πi/m)
5          for k ← 0 to n − 1 by m
6              do ω ← 1
7                  for j ← 0 to m/2 − 1
8                      do t ← ωA[k + j + m/2]
9                         u ← A[k + j]
10                        A[k + j] ← u + t
11                        A[k + j + m/2] ← u − t
12                        ω ← ωω_n
```

Si presenta ora la versione finale del codice iterativo per FFT che inverte i due cicli interni per eliminare qualche calcolo di indici e usa la procedura ausiliaria BIT-REVERSE-COPY (a, A) per copiare il vettore a nell'array A nell'opportuno ordine iniziale.

```
ITERATIVE-FFT(a)
1  BIT-REVERSE-COPY(a, A)
2  n ← length[a]              ▷ n è una potenza di 2
3  for s ← 1 to lg n
4  do    m ← 2^s
5        ω_m ← e^(2πi/m)
6        ω ← 1
7        for j ← 0 to m/2 − 1
```

```
8          do   for k ← j to n − 1 by m
9               do   t ← ω A[k + m/2]
10                   u ← A[k]
11                   A[k] ← u + t
12                   A[k + m/2] ← u-t
13              ω ← ωω_m
14 return A
```

Come fa la BIT-REVERSE-COPY a mettere gli elementi del vettore a di input nell'array A nell'ordine desiderato? L'ordine in cui le foglie appaiono nella figura 32.4 è "binario inverso". Cioè, sia rev(k) l'intero di lg$n$ bit formato dall'inversione dei bit della rappresentazione binaria di k; allora, si vuole mettere l'elemento a_k del vettore nella posizione dell'array A[rev(k)]. Nella figura 32.4, per esempio, le foglie appaiono nell'ordine 0, 4, 2, 6, 1, 5, 3, 7; questa sequenza in binario è 000, 100, 010, 110, 001, 101, 011, 111 e in binario inverso si ha la sequenza 000, 001, 010, 011, 100, 101, 110, 111. Per mostrare che si vuole in generale l'ordine binario inverso si noti che sul livello in alto dell'albero gli indici il cui bit meno significativo è 0 sono messi nel sottoalbero sinistro. Trascurando il bit meno significativo ad ogni livello, si continua questa procedura scendendo nell'albero, finché si ottiene l'ordine binario inverso delle foglie.

Poiché la funzione rev si calcola facilmente, la procedura BIT-REVERSE-COPY può essere scritta come segue.

```
BIT-REVERSE-COPY(a, A)
1  n ← length[a]
2  for k ← 0 to n − 1
3       do A[rev(k)] ← a_k
```

La realizzazione iterativa di FFT impiega tempo $\Theta(n \lg n)$. La chiamata a BIT-REVERSE-COPY(a, A) sicuramente impiega tempo $O(n \lg n)$ poiché si ripete n volte e si può invertire un intero compreso tra 0 ed $n - 1$ di lg$n$ bit in tempo $O(\lg n)$. (In pratica, di solito si conosce in anticipo il valore iniziale di n; sarebbe dunque accettabile codificare una tabella di corrispondenza tra k e rev(k), eseguendo BIT-REVERSE-COPY in tempo $\Theta(n)$ con una costante nascosta bassa. Alternativamente, si potrebbe usare lo schema astuto del contatore binario inverso descritto nel Problema 18-1.) Per completare la dimostrazione che ITERATIVE-FFT impiega tempo $\Theta(n \lg n)$, si mostra che $L(n)$, il numero di volte in cui il corpo del ciclo più interno (linee 9-12) è eseguito, è $\Theta(n \lg n)$. Si ha

$$
\begin{aligned}
L(n) &= \sum_{s=1}^{\lg n} \sum_{j=0}^{2^{s-1}-1} \frac{n}{2^s} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\
&= \sum_{s=1}^{\lg n} \frac{n}{2} \\
&= \Theta(n \lg n) .
\end{aligned}
$$

Figura 32.5 *Un circuito parallelo* PARALLEL-FFT *che calcola la FFT per $n = 8$ input. Gli stadi di farfalle sono etichettati per mostrare la corrispondenza con l'iterazione del ciclo più esterno della procedura* FFT-BASE. *Una FFT su n input può essere calcolata in profondità $\Theta(\lg n)$ con $\Theta(n \lg n)$ elementi combinatori.*

Un circuito parallelo per la FFT

Molte delle proprietà che consentono di realizzare un algoritmo iterativo per la FFT possono essere impiegate per produrre un algoritmo parallelo efficiente. (Si veda il Capitolo 29 per una descrizione del modello dei circuiti combinatori.) Il circuito combinatorio PARALLEL-FFT che calcola la FFT su n input è mostrato in figura 32.5 per $n = 8$. Il circuito comincia con una permutazione binaria inversa degli input, seguito da lg$n$ stadi, ognuno consistente di $n/2$ farfalle eseguite in parallelo. La profondità del circuito è allora $\Theta(\lg n)$.

La parte più a sinistra del circuito PARALLEL-FFT esegue la permutazione binaria inversa, e la parte restante simula la procedura iterativa FFT-BASE. Si trae vantaggio dal fatto che ogni iterazione del ciclo **for** più esterno esegue $n/2$ operazioni a farfalla indipendenti che possono essere eseguite in parallelo. Il valore di s in ogni iterazione in FFT-BASE corrisponde a uno stadio di farfalle mostrato nella figura 32.5. Nello stadio s, per $s = 1, 2, \ldots, \lg n$, vi sono $n/2^s$ gruppi di farfalle (uno per ogni valore di k nella FFT-BASE), con 2^{s-1} farfalle per gruppo (una per ogni valore di j nella FFT-BASE). Le farfalle mostrate in figura 32.5 corrispondono alle operazioni a farfalla del ciclo più interno (linee 8-11 della FFT-BASE). Si noti anche che i valori di ω usati nelle farfalle corrispondono a quelli usati nella FFT-BASE: nello stadio s si usa $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, dove $m = 2^s$.

Esercizi

32.3-1 Si mostri come ITERATIVE-FFT calcola la DFT del vettore di input (0, 2, 3, −1, 4, 5, 7, 9).

32.3-2 Si mostri come realizzare un algoritmo per la FFT dove la permutazione binaria inversa avviene alla fine piuttosto che all'inizio del calcolo. (*Suggerimento:* si consideri la DFT inversa.)

32.3-3 Nel calcolo di DFT_n, quanti elementi per addizione, sottrazione e moltiplicazione e quanti fili sono necessari nel circuito PARALLEL-FFT descritto in questo paragrafo? (Si assuma che sia necessario un solo filo per portare un numero da un posto all'altro.)

* *32.3-4* Si supponga che gli addizionatori nel circuito per la FFT talvolta si guastino in modo tale da produrre come output sempre zero, indipendentemente dai loro input. Si supponga che esattamente uno degli addizionatori sia guasto, ma di non sapere quale. Si descriva come identificare l'addizionatore guasto fornendo opportuni input all'intero circuito e osservandone gli output. Si descriva una procedura efficiente.

Problemi

32-1 Moltiplicazione divide-et-impera

a. Si mostri come moltiplicare due polinomi $ax + b$ e $cx + d$ di grado 1 usando solo tre moltiplicazioni (*Suggerimento:* una delle moltiplicazioni è $(a + b) \cdot (c + d)$.)

b. Si diano due algoritmi divide-et-impera per moltiplicare due polinomi di grado-limite n che impieghino tempo $\Theta(n^{\lg 3})$. Il primo algoritmo dovrebbe dividere i coefficienti dei polinomi di input in una metà alta e una metà bassa e il secondo algoritmo dovrebbe dividerli a seconda che i loro indici siano pari o dispari.

c. Si mostri che due interi di n bit possono essere moltiplicati in $O(n^{\lg 3})$ passi, dove ogni passo opera su al più un numero costante di bit a valore 1.

32-2 Matrici di Toeplitz

Una *matrice di Toeplitz* è una matrice $A = (a_{ij})$ $n \times n$ tale che $a_{ij} = a_{i-1,j-1}$ per $i = 2, 3, \ldots, n$ e $j = 2, 3, \ldots, n$.

a. La somma di due matrici di Toeplitz è necessariamente una matrice di Toeplitz? E il prodotto?

b. Si descriva come rappresentare una matrice di Toeplitz così che due matrici di Toeplitz $n \times n$ possano essere sommate in tempo $O(n)$.

c. Si dia un algoritmo che richiede tempo $O(n \lg n)$ per moltiplicare una matrice di Toeplitz per un vettore di lunghezza n. Si usi la rappresentazione trovata per la parte (b).

d. Si dia un algoritmo efficiente per moltiplicare due matrici di Toeplitz $n \times n$. Si analizzi il suo tempo di esecuzione.

32-3 Valutazione di tutte le derivate di un polinomio in un punto

Dato un polinomio $A(x)$ di grado-limite n, la sua derivata t-esima è definita da

$$A^{(t)}(x) = \begin{cases} A(x) & \text{se } t = 0 , \\ \frac{d}{dx} A^{(t-1)}(x) & \text{se } 1 \le t \le n-1 , \\ 0 & \text{se } t \ge n . \end{cases}$$

A partire dalla rappresentazione a coefficienti $(a_0, a_1, \ldots, a_{n-1})$ di $A(x)$ e da un dato punto x_0, si desidera determinare $A^{(t)}(x_0)$ per $t = 0, 1, \ldots, n-1$.

a. Dati i coefficienti $b_0, b_1, \ldots, b_{n-1}$ tali che

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j ,$$

si mostri come calcolare $A^{(t)}(x_0)$ per $t = 0, 1, \ldots, n-1$, in tempo $O(n)$.

b. Si spieghi come trovare $b_0, b_1, \ldots, b_{n-1}$ in tempo $O(n \lg n)$, dato $A(x_0 + \omega_n^k)$ per $k = 0, 1, \ldots, n-1$.

c. Si provi che

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right) ,$$

dove $f(j) = a_j j!$ e

$$g(l) = \begin{cases} x_0^{-l} / (-l)! & \text{se } -(n-1) \le l \le 0 , \\ 0 & \text{se } 1 \le l \le (n-1) . \end{cases}$$

d. Si spieghi come valutare $A(x_0 + \omega_n^k)$ per $k = 0, 1, \ldots, n-1$, in tempo $O(n \lg n)$. Si concluda che tutte le derivate non banali di $A(x)$ possono essere valutate nel punto x_0 in tempo $O(n \lg n)$.

32-4 Valutazione di polinomi in più punti

Si è osservato che il problema della valutazione di polinomi di grado-limite $n - 1$ in un solo punto può essere risolto in tempo $O(n)$ usando la regola di Horner. Si è anche scoperto che tali polinomi possono essere valutati in tutte le n radici complesse dell'unità in tempo $O(n \lg n)$ usando la FFT. Si vedrà ora che valutare un polinomio di grado-limite n in n punti arbitrari in tempo $O(n \lg^2 n)$.

Per far ciò, si userà il fatto che è possibile calcolare in tempo $O(n \lg n)$ il resto della divisione di un polinomio di grado limite n per un altro polinomio, risultato che si assume senza dimostrazione. Per esempio, il resto della divisione di $3x^3 + x^2 - 3x + 1$ per $x^2 + x + 2$ è

$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = 7x + 5$.

Data la rappresentazione a coefficienti di un polinomio $A(x) = \sum_{k=0}^{n-1} a_k x^k$ e dati n punti $x_0, x_1, \ldots, x_{n-1}$, si desidera calcolare gli n valori $A(x_0), A(x_1), \ldots, A(x_{n-1})$. Per $0 \le i \le j \le n-1$, si definiscono i polinomi $P_{ij}(x) = \prod_{k=i}^{j}(x - x_k)$ e $Q_{ij}(x) = A(x) \bmod p_{ij}(x)$. Si noti che $Q_{ij}(x)$ ha grado-limite al più $j - i$.

a. Si dimostri che $A(x) \bmod (x - z) = A(z)$ per qualunque punto z.

b. Si dimostri che $Q_{kk}(x) = A(x_k)$ e che $Q_{0,n-1}(x) = A(x)$.

c. Si dimostri che, per $i \le k \le j$, si ha $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ e $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

d. Si dia un algoritmo di tempo $O(n \lg^2 n)$ per valutare $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

32-5 *FFT con aritmetica modulare*

Per come è stata definita, la Trasformata Discreta di Fourier richiede l'uso di numeri complessi, il che può talvolta indurre a una mancanza di precisione dovuta a errori di arrotondamento. Per alcuni problemi, si sa anticipatamente che la soluzione contiene solo interi ed è preferibile utilizzare una variante della FFT basata sull'aritmetica modulare al fine di garantire che la soluzione sia calcolata in modo preciso. Un esempio di tali problemi è la moltiplicazione di due polinomi con coefficienti interi. L'Esercizio 32.2-6 fornisce un approccio che usa un modulo di $\Omega(n)$ bit per gestire una DFT su n punti. Questo problema fornisce un altro approccio che usa un modulo di lunghezza più ragionevole $O(\lg n)$: è richiesta la conoscenza del materiale al Capitolo 33. Sia n una potenza di 2.

a. Si supponga di cercare il più piccolo k tale che $p = kn + 1$ sia primo. Si dia un semplice motivo euristico per cui ci si dovrebbe aspettare che k sia approssimativamente $\lg n$. (Il valore di k potrebbe essere più grande o più piccolo, ma è ragionevole aspettarsi di esaminare in media $O(\lg n)$ valori candidati per k.) Quanto è paragonabile la lunghezza media di p con la lunghezza di n?

Sia g un generatore di $\mathbf{Z}_p^*$ e sia $w = g^k \bmod p$.

b. Si deduca che la DFT e la DFT inversa sono operazioni modulo p ben definite, dove w è usata come radice primitiva n-esima dell'unità.

c. Si deduca che la FFT e la sua inversa possono funzionare modulo p in tempo $O(n \lg n)$, dove le operazioni sulle parole di $O(\lg n)$ bit richiedono tempo unitario. Si assuma che siano dati p e w.

d. Si calcoli la DFT modulo $p = 17$ del vettore $(0, 5, 3, 7, 7, 2, 1, 6)$. Si noti che $g = 3$ è un generatore di $\mathbf{Z}_{17}^*$.

Note al capitolo

Press, Flannery, Teukolsky e Vetterling [161, 162] danno una buona descrizione della Trasformata Veloce di Fourier e delle sue applicazioni. Per un ottima introduzione alla elaborazione dei segnali, un'area molto conosciuta di applicazione delle FFT, si veda il testo di Oppenheim e Willsky [153].

A Cooley e Tukey [51] è stata attribuita l'ideazione della FFT negli anni 60. In effetti la FFT è stata scoperta molto tempo prima, ma la sua importanza non fu compresa pienamente prima dell'avvento dei moderni calcolatori numerici. Press, Flannery, Teukolsky e Vetterling attribuiscono le origini del metodo a Runge e König (1924).

Algoritmi di teoria dei numeri

33

La teoria dei numeri un tempo era vista come un argomento della matematica pura, bello ma assolutamente inutile. Algoritmi di teoria dei numeri sono ampiamente usati al giorno d'oggi, grazie in parte all'invenzione degli schemi crittografici basati su grandi numeri primi. La praticabilità di questi schemi sta nella capacità di trovare facilmente i numeri primi, mentre la loro sicurezza sta nell'incapacità di scomporre in fattori il prodotto di grandi numeri primi. Questo capitolo presenta alcuni aspetti della teoria dei numeri e gli algoritmi associati che sono alla base di tali applicazioni.

Il paragrafo 33.1 introduce i concetti di base della teoria dei numeri, come ad esempio la divisibilità, la congruenza modulare e l'unicità della scomposizione in fattori. Il paragrafo 33.2 studia uno degli algoritmi più vecchi del mondo: l'algoritmo di Euclide per calcolare il massimo comun divisore di due interi. Il paragrafo 33.3 passa in rassegna i concetti dell'aritmetica modulare. Il paragrafo 33.4 studia l'insieme dei multipli di un dato numero a modulo n e mostra come trovare tutte le soluzioni dell'equazione $ax \equiv b \pmod{n}$ usando l'algoritmo di Euclide. Il teorema del resto cinese è presentato nel paragrafo 33.5. Il paragrafo 33.6 considera le potenze di un dato numero a modulo n e presenta un algoritmo con quadrature ripetute per calcolare in modo efficiente $a^b \bmod n$, dati a, b e n. Questa operazione è centrale per verificare efficientemente la primalità. Il paragrafo 33.7 descrive poi il sistema di crittografia a chiave pubblica RSA. Il paragrafo 33.8 descrive una verifica di primalità randomizzata che può essere usata in modo efficiente per trovare grandi numeri primi, un aspetto essenziale della creazione delle chiavi nella crittografia RSA. Infine, il paragrafo 33.9 presenta una semplice ma efficace euristica per la scomposizione in fattori di interi piccoli. È curioso il fatto che la scomposizione in fattori sia un problema che la gente auspicherebbe fosse intrattabile, ciò in virtù della considerazione che la sicurezza dell'RSA dipende dalla difficoltà di scomporre in fattori interi grandi.

Dimensione dell'input e costo dei calcoli aritmetici

Poiché si lavorerà con interi grandi, è necessario rivedere i concetti di dimensione dell'input e di costo delle operazioni aritmetiche elementari.

In questo capitolo, un "input grande" tipicamente significa un input contenente "interi grandi" piuttosto che un input contenente "molti interi" (come per l'ordinamento). Pertanto, si misurerà la dimensione di un input in termini del numero di bit richiesti per rappresentare quell'input e non del numero di interi dell'input. Un algoritmo con input interi $a_1, a_2, \ldots, a_k$

è un algoritmo di ***tempo polinomiale*** se impiega tempo polinomiale rispetto a $\lg a_1, \lg a_2, \ldots, \lg a_k$, cioè polinomiale rispetto alle lunghezze dei suoi input codificati in binario.

In genere, in questo libro si è assunto per comodità che le operazioni aritmetiche elementari (moltiplicazione, divisione, calcolo del resto) fossero operazioni primitive che richiedono un'unità di tempo. Contando il numero di tali operazioni aritmetiche che un algoritmo esegue, si ha una base su cui fare una stima ragionevole del tempo reale di esecuzione di un algoritmo su un calcolatore. Le operazioni elementari però, quando i loro input sono grandi, possono richiedere un tempo maggiore. Pertanto diventa conveniente misurare quante ***operazioni sui bit*** richiede un algoritmo di teoria dei numeri. In questo modello, una moltiplicazione di due interi di β bit con il metodo comune richiede $\Theta(\beta^2)$ operazioni sui bit. Analogamente, l'operazione di divisione di un intero di β bit per un intero più corto, o l'operazione di prendere il resto della divisione di un numero di β bit per un intero più corto, possono essere eseguite in tempo $\Theta(\beta^2)$ con algoritmi semplici. (Si veda l'Esercizio 33.1-11.) Sono noti metodi più veloci. Per esempio un semplice metodo divide-et-impera per moltiplicare due interi di β bit ha tempo di esecuzione $\Theta(\beta \lg \beta \lg \lg \beta)$. Tuttavia ai fini pratici l'algoritmo $\Theta(\beta^2)$ è spesso migliore, e si userà questo limite come base per le analisi degli algoritmi.

In questo capitolo gli algoritmi sono generalmente analizzati in termini del numero di operazioni aritmetiche e di operazioni sui bit richieste.

33.1 Nozioni elementari di teoria dei numeri

Questo paragrafo fornisce una breve presentazione delle nozioni elementari di teoria dei numeri che riguarda l'insieme $\mathbf{Z} = \{\ldots -2, -1, 0, 1, 2, \ldots\}$ degli interi e l'insieme $\mathbf{N} = \{0, 1, 2, \ldots\}$ dei naturali.

Divisibilità e divisori

La nozione di divisibilità di un intero per un altro è centrale nella teoria dei numeri. La notazione $d|a$ (si legga "*d divide a*") significa che $a = kd$ per qualche intero k. Ogni intero divide 0. Se $a > 0$ e $d \mid a$, allora $|d| \le |a|$. Se $d \mid a$, allora si dice anche che a è ***multiplo*** di d. Se d non divide a allora si scrive $d \nmid a$.

Se $d \mid a$ e $d \ge 0$, si dice che d è un ***divisore*** di a. Si noti che $d|a$ se e solo se $-d \mid a$, così che non si perde generalità se si definiscono i divisori come numeri non negativi, restando sottinteso che anche il negativo di qualunque divisore di a divide a. Un divisore di un intero a è compreso tra 1 e $|a|$. Per esempio, i divisori di 24 sono 1, 2, 3, 4, 6, 8, 12 e 24. Ogni intero a è divisibile per i ***divisori banali*** 1 e a. I divisori non banali di a sono anche chiamati ***fattori*** di a. Per esempio, i fattori di 20 sono 2, 4, 5 e 10.

Numeri primi e composti

Un intero $a > 1$ i cui unici divisori sono quelli banali 1 e a si dice ***numero primo*** (o, più semplicemente, ***primo***). I primi hanno molte proprietà speciali e ricoprono un ruolo critico nella teoria dei numeri. I primi piccoli sono, nell'ordine,

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, ...

L'Esercizio 33.1-1 richiede di dimostrare che vi sono infiniti primi. Un intero $a > 1$ che non è primo è detto ***numero composto*** (o, più semplicemente, ***composto***). Per esempio, 39 è composto perché $3|39$. L'intero 1 è detto ***unità*** e non è né primo né composto. Analogamente, l'intero 0 e tutti gli interi negativi non sono né primi né composti.

Il teorema di divisione, dei resti e di congruenza modulare

Dato un numero n, gli interi possono essere partizionati in quelli che sono multipli di n e quelli che non sono multipli di n. Molti aspetti della teoria dei numeri sono basati su un raffinamento di questa partizione ottenuta classificando i non-multipli di n a seconda del resto che si ottiene dividendoli per n. Il seguente teorema è la base di questo raffinamento. La dimostrazione di questo teorema non sarà data (si veda ad esempio, Niven e Zuckerman [151]).

Teorema 33.1 (Teorema della divisione)

Per qualsiasi intero a e qualunque intero positivo n, vi sono due unici interi q e r tali che $0 \le r < n$ e $a = qn + r$. ■

Il valore $q = \lfloor a/n \rfloor$ è il ***quoziente*** della divisione. Il valore $r = a \bmod n$ è il ***resto*** della divisione. Si ha che $n|a$ se e solo se $a \bmod n = 0$. Segue che

$$a = \lfloor a/n \rfloor n + (a \bmod n) \tag{33.1}$$

oppure

$$a \bmod n = a - \lfloor a/n \rfloor n . \tag{33.2}$$

Data una nozione ben definita di resto della divisione di un intero per un altro, è conveniente fornire una notazione particolare per indicare l'uguaglianza dei resti. Se $(a \bmod n) = (b \bmod n)$, si scrive $a \equiv b \pmod n$ e si dice che a è ***congruente*** a b modulo n. In altre parole $a \equiv b \pmod n$ se a e b danno lo stesso resto quando sono divisi per n. Allo stesso modo, $a \equiv b \pmod n$ se e solo se $n|(b - a)$. Si scrive $a \not\equiv b \pmod n$ se a non è congruente a b modulo n. Per esempio, $61 \equiv 6 \pmod{11}$. Inoltre $-13 \equiv 22 \equiv 2 \pmod 5$.

Gli interi possono essere divisi in n classi di equivalenza a seconda del resto modulo n. La ***classe di congruenza modulo n*** contenente un intero a è la classe di equivalenza

$$[a]_n = \{a + kn : k \in \mathbf{Z}\}$$

Per esempio, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$; altre denotazioni per questo insieme sono $[-4]_7$ e $[10]_7$. Scrivere $a \in [b]_n$ è come scrivere $a \equiv b \pmod n$. L'insieme di tutte le classi di equivalenza è

$$\mathbf{Z}_n = \{[a]_n : 0 \le a \le n - 1\} . \tag{33.3}$$

La definizione

$$\mathbf{Z}_n = \{0, 1, \ldots, n - 1\} , \tag{33.4}$$

dovrebbe essere letta come equivalente alla (33.3) restando sottinteso che 0 rappresenta $[0]_n$, 1 rappresenta $[1]_n$ e così via: ogni classe è rappresentata dal suo elemento non negativo

minimo. Tuttavia, deve però essere tenuta in considerazione la classe di equivalenza sottostante. Per esempio, un riferimento a -1 come membro della classe di $\mathbf{Z}_n$ è un riferimento alla classe $[n-1]_n$, poiché $-1 \equiv n-1 \pmod{n}$.

Divisori comuni e massimo comun divisore

Se d è un divisore di a e anche un divisore di b, allora d è un ***divisore comune*** di a e b. Per esempio, i divisori di 30 sono 1, 2, 3, 5, 6, 10, 15 e 30, così i divisori comuni di 24 e 30 sono 1, 2, 3 e 6. Si noti che 1 è un divisore comune di due interi qualsiasi.

Una proprietà importante dei divisori comuni è che

$$d \mid a \text{ e } d \mid b \text{ implica } d \mid (a+b) \text{ e } d \mid (a-b) . \tag{33.5}$$

Più generalmente si ha che

$$d \mid a \text{ e } d \mid b \text{ implica } d \mid (ax+by) \tag{33.6}$$

per x e y interi qualsiasi. Inoltre, se $a \mid b$, allora $|a| \le |b|$ oppure $b = 0$ che implica che

$$a \mid b \text{ e } b \mid a \text{ implica } a = \pm b. \tag{33.7}$$

Il ***massimo comun divisore*** di due interi a e b, non entrambi 0, è il più grande dei divisori comuni di a e b; è denotato da $\mathrm{MCD}(a,b)$. Per esempio $\mathrm{MCD}(24,30) = 6$, $\mathrm{MCD}(5,7) = 1$ e $\mathrm{MCD}(0,9) = 9$. Se a e b non sono entrambi 0, allora $\mathrm{MCD}(a,b)$ è un intero tra 1 e $\min(|a|,|b|)$. Si definisce $\mathrm{MCD}(0,0) = 0$; questa definizione è necessaria per rendere le proprietà standard della funzione MCD (come ad esempio l'equazione (33.11) che segue) universalmente valide.

Le seguenti sono le proprietà elementari della funzione MCD:

$$\mathrm{MCD}(a,b) = \mathrm{MCD}(b,a) , \tag{33.8}$$

$$\mathrm{MCD}(a,b) = \mathrm{MCD}(-a,b) , \tag{33.9}$$

$$\mathrm{MCD}(a,b) = \mathrm{MCD}(|a|,|b|) , \tag{33.10}$$

$$\mathrm{MCD}(a,0) = |a| , \tag{33.11}$$

$$\mathrm{MCD}(a,ka) = |a| \text{ per qualunque } k \in \mathbf{Z} . \tag{33.12}$$

Teorema 33.2

Se a e b sono interi qualsiasi, non entrambi zero, allora il $\mathrm{MCD}(a,b)$ è il più piccolo elemento positivo dell'insieme $\{ax + by : x,y \in \mathbf{Z}\}$ di combinazioni lineari di a e b.

Dimostrazione. Sia s il più piccolo positivo combinazione lineare di a e b, e sia $s = ax + by$ per qualche $x, y \in \mathbf{Z}$. Sia $q = \lfloor a/s \rfloor$. L'equazione (33.2) allora implica

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a(1 - qx) + b(-qy) , \end{aligned}$$

e pertanto anche $a \bmod s$ è una combinazione lineare di a e b. Ma, poiché $a \bmod s < s$, si ha che $a \bmod s = 0$ perché s è il più piccolo positivo che è combinazione lineare. Allora, $s|a$ e,

per ragioni analoghe, $s|b$. Pertanto s è un divisore comune di a e b e così $\mathrm{MCD}(a,b) \ge s$. L'equazione (33.6) implica che $\mathrm{MCD}(a,b) | s$, poiché $\mathrm{MCD}(a,b)$ divide sia a che b ed s è una combinazione lineare di a e b. Ma $\mathrm{MCD}(a,b) | s$ e $s > 0$ implica che $\mathrm{MCD}(a,b) \le s$. combinando $\mathrm{MCD}(a,b) \ge s$ e $\mathrm{MCD}(a,b) \le s$ si ottiene $\mathrm{MCD}(a,b) = s$; si conclude che s è il massimo comun divisore di a e b. ■

Corollario 33.3

Per due interi a e b qualsiasi, se $d|a$ e $d|b$ allora $d \mid \mathrm{MCD}(a,b)$.

Dimostrazione. Questo corollario segue dall'equazione (33.6), essendo $\mathrm{MCD}(a,b)$ una combinazione lineare di a e b per il Teorema 33.2. ■

Corollario 33.4

Per tutti gli interi a e b e qualunque intero n non negativo,
$\mathrm{MCD}(an,bn) = n\mathrm{MCD}(a,b)$.

Dimostrazione. Se $n = 0$, il corollario è banale. Se $n > 0$ allora $\mathrm{MCD}(an, bn)$ è l'elemento positivo più piccolo dell'insieme $\{anx + bny\}$ che è n volte l'elemento positivo più piccolo dell'insieme $\{ax + by\}$. ■

Corollario 33.5

Per tutti gli interi n, a e b, se $n|ab$ e $\mathrm{MCD}(a,n) = 1$, allora $n|b$.

Dimostrazione. La dimostrazione è lasciata come Esercizio 33.1-4. ■

Interi primi tra loro

Due interi a, b sono detti ***primi tra loro*** se il loro unico comune divisore è 1 cioè se $\mathrm{MCD}(a,b) = 1$. Per esempio, 8 e 15 sono primi tra loro poiché i divisori di 8 sono 1, 2, 4 e 8, mentre i divisori di 15 sono 1, 3, 5 e 15. Il seguente teorema stabilisce che se due interi sono tali che ognuno di essi e l'intero p sono primi tra loro, allora il loro prodotto e p sono primi tra loro.

Teorema 33.6

Per qualsiasi intero a, b e p se $\mathrm{MCD}(a, p) = 1$ e $\mathrm{MCD}(b, p) = 1$, allora $\mathrm{MCD}(ab, p) = 1$.

Dimostrazione. Dal Teorema 33.2 segue che esistono gli interi x, y, x' e y' tali che

$$ax + py = 1,$$
$$bx' + py' = 1.$$

Moltiplicando queste equazioni e risistemandole si ha

$$ab(xx') + p(ybx' + y'ax + pyy') = 1 .$$

Poiché 1 è così una combinazione lineare di ab e p, applicando il Teorema 33.2 si completa la prova. ■

Si dice che gli interi $n_1, n_2, \ldots, n_k$ sono ***primi tra loro a coppie*** se, per $i \neq j$, si ha MCD $(n_i, n_j) = 1$.

Unicità della scomposizione in fattori primi

Una proprietà elementare ma importante sulla divisibilità per numeri primi è la seguente

Teorema 33.7

Per tutti i primi p e tutti gli interi a, b se $p|ab$ allora $p|a$ o $p|b$.

Dimostrazione. Si supponga per assurdo che $p|ab$ ma che $p \nmid a$ e $p \nmid b$. Pertanto, MCD$(a, p) = 1$ e MCD$(b, p) = 1$, poiché gli unici divisori di p sono 1 e p e, per ipotesi, p non divide né a né b. Allora il Teorema 33.6 implica che MCD$(ab,p) = 1$, contraddicendo l'ipotesi che $p|ab$, poiché $p|ab$ implica MCD$(ab, p) = p$.
Questa contraddizione completa la dimostrazione. ■

Una conseguenza del Teorema 33.7 è che un intero ha un'unica scomposizione in fattori primi.

Teorema 33.8 (Unicità della scomposizione in fattori primi)

Un intero composto a può essere scritto come un prodotto della forma

$$a = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$$

in un unico modo, dove i p_i sono numeri primi, $p_1 < p_2 < \ldots < p_r$, e gli e_i sono interi positivi.

Dimostrazione. La dimostrazione è lasciata come Esercizio 33.1-10. ■

Per esempio, il numero 6000 può essere scomposto in fattori primi in modo unico come $2^4 \cdot 3 \cdot 5^3$.

Esercizi

33.1-1 Si dimostri che vi sono infiniti primi. (*Suggerimento:* si mostri che nessuno dei primi $p_1, p_2, \ldots, p_k$ divide $(p_1 p_2 \ldots p_k) + 1$.)

33.1-2 Si dimostri che se $a|b$ e $b|c$, allora $a|c$.

33.1-3 Si dimostri che se p è primo e $0 < k < p$, allora MCD$(k, p) = 1$.

33.1-4 Si dimostri il Corollario 33.5.

33.1-5 Si dimostri che se p è primo e $0 < k < p$, allora $p \mid \binom{p}{k}$. Si concluda che per tutti gli interi a, b, e tutti i numeri primi p,
$(a + b)^p \equiv a^p + b^p \pmod{p}$.

33.1-6 Si dimostri che per due interi qualsiasi a e b, tali che $a|b$ e $b > 0$,
$(x \bmod b) \bmod a = x \bmod a$
per qualunque x. Si dimostri, sotto le stesse ipotesi, che
$x \equiv y \pmod{b}$ implica $x \equiv y \pmod{a}$
per qualsiasi x e y interi.

33.1-7 Per qualunque intero $k > 0$, si dice che un intero n è una ***k-esima potenza*** se esiste un intero a tale che $a^k = n$. Si dice che $n > 1$ è una ***potenza non banale*** se è una k-esima potenza per qualche intero $k > 1$. Si mostri come determinare se un dato intero n di β bit è una potenza non banale in tempo polinomiale in β.

33.1-8 Si dimostrino le equazioni (33.8)-(33.12).

33.1-9 Si mostri che l'operatore MCD è associativo. Cioè si dimostri che per tutti gli interi a, b e c vale
MCD$(a,$ MCD$(b,c)) =$ MCD(MCD$(a,b),c)$.

* ***33.1-10*** Si dimostri il Teorema 33.8.

33.1-11 Si dia un algoritmo efficiente per l'operazione di divisione di un intero di β bit per un intero più corto e l'operazione di resto della divisione di un intero di β bit per un intero più corto. L'algoritmo dovrebbe impiegare tempo $O(\beta^2)$.

33.1-12 Si dia un algoritmo efficiente per convertire un dato intero (binario) di β bit nella sua rappresentazione decimale. Si deduca che se la moltiplicazione o la divisione di interi la cui lunghezza è al più β richiede $M(\beta)$, allora la conversione da binario a decimale può essere eseguita in tempo $\Theta(M(\beta) \lg \beta)$. (*Suggerimento:* si usi un approccio divide-et-impera, ottenendo le due metà del risultato con ricorsioni separate.)

33.2 Massimo comun divisore

In questo paragrafo, si usa l'algoritmo di Euclide per calcolare il massimo comun divisore di due interi. L'analisi del tempo di esecuzione porta a una sorprendente connessione con i numeri di Fibonacci che rappresentano l'input peggiore per l'algoritmo di Euclide.

In questo paragrafo ci si limita a interi non negativi. Questa restrizione è giustificata dall'equazione (33.10) che asserisce che MCD$(a, b) =$ MCD$(|a|, |b|)$.

Generalmente, si può calcolare il MCD(a,b) per due interi a e b positivi dalla scomposizione in fattori primi di a e b. In effetti, se

$$a = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r} \quad , \tag{33.13}$$

$$b = p_1^{f_1} p_2^{f_2} \ldots p_r^{f_r} \quad , \tag{33.14}$$

con esponenti eventualmente uguali a zero in modo da usare lo stesso insieme di primi $p_1, p_2, \ldots, p_r$ per a e b, allora

$$\mathrm{MCD}(a,b) = p_1^{\min(e_1,f_1)} p_2^{\min(e_2,f_2)} \cdots p_r^{\min(e_r,f_r)} . \tag{33.15}$$

Come si vedrà nel paragrafo 33.9, finora i migliori algoritmi per la scomposizione in fattori non vengono eseguiti in tempo polinomiale. Pertanto, questo approccio per calcolare il massimo comun divisore non sembra portare ad un algoritmo efficiente.

L'algoritmo di Euclide per il calcolo del massimo comun divisore è basato sul seguente teorema.

Teorema 33.9 (Teorema di ricorsione del MCD)

Per qualunque intero a non negativo e qualunque intero b positivo,

$\mathrm{MCD}(a, b) = \mathrm{MCD}(b, a \bmod b)$.

Dimostrazione. Si mostrerà che $\mathrm{MCD}(a,b)$ e $\mathrm{MCD}(b,a \bmod b)$ si dividono l'un l'altro, quindi per l'equazione (33.7) devono essere uguali (poiché sono entrambi non negativi).

Prima si mostra che $\mathrm{MCD}(a, b) | \mathrm{MCD}(b, a \bmod b)$. Se si pone $d = \mathrm{MCD}(a, b)$, allora $d|a$ e $d|b$. Dall'equazione (33.2), $(a \bmod b) = a - qb$, dove $q = \lfloor a/b \rfloor$. Quindi, poiché $(a \bmod b)$ è una combinazione lineare di a e b, l'equazione (33.6) implica che $d|(a \bmod b)$. Allora, poiché $d|b$ e $d|(a \bmod b)$, il corollario 33.3 implica che $d|\mathrm{MCD}(b,a \bmod b)$, o equivalentemente, che

$$\mathrm{MCD}(a, b) \mid \mathrm{MCD}(b, a \bmod b). \tag{33.16}$$

La dimostrazione che $\mathrm{MCD}(b,a \bmod b)|\mathrm{MCD}(a,b)$ è molto simile. Se si pone ora $d = \mathrm{MCD}(b, a \bmod b)$, allora $d|b$ e $d|(a \bmod b)$. Poiché $a = qb + (a \bmod b)$, dove $q = \lfloor a/b \rfloor$, si ha che a è una combinazione lineare di b e di $(a \bmod b)$. Dall'equazione (33.6), si conclude che $d|a$. Poiché $d|b$ e $d|a$, si ha che $d|\mathrm{MCD}(a,b)$ dal Corollario 33.3 o equivalentemente che

$$\mathrm{MCD}(b, a \bmod b) \mid \mathrm{MCD}(a, b). \tag{33.17}$$

Usando l'equazione (33.7) per combinare le equazioni (33.16) e (33.17) si completa la prova. ■

Algoritmo di Euclide

Il seguente algoritmo per il MCD è descritto negli *Elementi* di Euclide (*circa* 300 a.C.), sebbene potrebbe essere di origini ancora più vecchie. È scritto come un programma ricorsivo basato direttamente sul Teorema 33.9. Gli input a e b sono interi non negativi qualsiasi.

```
EUCLID(a, b)
1  if b = 0
2      then return a
3      else return EUCLID(b, a mod b)
```

Quale esempio di esecuzione dell'algoritmo di Euclide, si consideri il calcolo di MCD(30, 21):

$$\begin{aligned} \text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\ &= \text{EUCLID}(9, 3) \\ &= \text{EUCLID}(3, 0) \\ &= 3. \end{aligned}$$

In questo calcolo, vi sono tre chiamate ricorsive di EUCLID.

La correttezza di EUCLID segue dal Teorema 33.9 e dal fatto che, se l'algoritmo restituisce a alla linea 2, allora $b = 0$, come l'equazione (33.11) implica che $\mathrm{MCD}(a,b) = \mathrm{MCD}(a,0) = a$. L'algoritmo non può richiamare sé stesso all'infinito perché il secondo argomento decresce in modo stretto ad ogni chiamata ricorsiva. Quindi EUCLID termina sempre con la soluzione corretta.

Tempo di esecuzione dell'algoritmo di Euclide

Si analizza ora il tempo di esecuzione di EUCLID nel caso peggiore in funzione della grandezza di a e b. Si assume, con una piccola perdita di generalità, che $a > b > 0$. Questa ipotesi può essere giustificata dall'osservazione che se $b > a \geq 0$, allora EUCLID(a,b) richiama immediatamente EUCLID(b,a). Cioè, se il primo argomento è minore del secondo, EUCLID usa una chiamata ricorsiva per scambiare i suoi argomenti e poi procede. Analogamente, se $b = a > 0$, la procedura termina dopo una chiamata ricorsiva, poiché $a \bmod b = 0$.

Il tempo di esecuzione complessivo di EUCLID è proporzionale al numero di chiamate ricorsive richieste. L'analisi utilizza i numeri di Fibonacci F_k, definiti dalla ricorrenza (2.13).

Lemma 33.10
Se $a > b \geq 0$ e l'invocazione di EUCLID (a, b) esegue $k \geq 1$ chiamate ricorsive, allora $a \geq F_{k+2}$ e $b \geq F_{k+1}$.

Dimostrazione. La dimostrazione è per induzione su k. Per la base dell'induzione, sia $k = 1$. Allora, $b \geq 1 = F_2$ e poiché $a > b$, si deve avere $a \geq 2 = F_3$. Poiché $b > (a \bmod b)$, in ogni chiamata ricorsiva il primo argomento è strettamente più grande del secondo; l'ipotesi $a > b$, allora, vale per ogni chiamata ricorsiva.

Si assuma induttivamente che il lemma sia vero se vengono fatte $k - 1$ chiamate ricorsive; si proverà allora il lemma per k chiamate ricorsive. Poiché $k > 0$, si ha $b > 0$ e EUCLID(a, b) chiama ricorsivamente EUCLID $(b, a \bmod b)$ che a sua volta esegue $k - 1$ chiamate ricorsive. L'ipotesi induttiva allora implica che $b \geq F_{k+1}$ (provando così parte del lemma), e $(a \bmod b) \geq F_k$. Si ha

$$\begin{aligned} b + (a \bmod b) &= b + (a - \lfloor a/b \rfloor b) \\ &\leq a , \end{aligned}$$

poiché $a > b > 0$ implica $\lfloor a/b \rfloor \geq 1$. Pertanto,

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2} . \end{aligned}$$

■

Il seguente teorema è un corollario immediato di questo lemma.

Teorema 33.11 (Teorema di Lamé)

Per qualunque intero $k \geq 1$, se $a > b \geq 0$ e $b < F_{k+1}$, allora l'invocazione EUCLID(a,b) esegue meno di k chiamate ricorsive. ■

Si può dimostrare che il limite superiore stabilito dal Teorema 33.11 è il migliore possibile. Numeri consecutivi di Fibonacci sono gli input peggiori per EUCLID. Poiché EUCLID(F_3, F_2) fa esattamente una chiamata ricorsiva e poiché per $k \geq 2$ si ha $F_{k+1} \bmod F_k = F_{k-1}$, si ha anche

$$\begin{aligned} \text{MCD}(F_{k+1}, F_k) &= \text{MCD}(F_k, (F_{k+1} \bmod F_k)) \\ &= \text{MCD}(F_k, F_{k-1}) . \end{aligned}$$

Pertanto, EUCLID(F_{k+1}, F_k) richiama sé stessa *esattamente* $k - 1$ volte, ricadendo nel limite superiore del Teorema 33.11.

Poiché F_k è approssimativamente $\phi^k/\sqrt{5}$, dove ϕ è il rapporto aureo $(1 + \sqrt{5})/2$ definito dall'equazione (2.14), il numero di chiamate ricorsive in EUCLID è $O(\lg b)$. (Si veda l'Esercizio 33.2-5 per un limite più stretto.) Segue che se Euclide è applicato a due numeri di β bit, allora esegue $O(\beta)$ operazioni aritmetiche e $O(\beta^3)$ operazioni sui bit (assumendo che la moltiplicazione e la divisione di numeri di β bit richiedano $O(\beta^2)$ operazioni sui bit). Il Problema 33-2 richiede di mostrare un limite $O(\beta^2)$ sul numero di operazioni sui bit.

Forma estesa dell'algoritmo di Euclide

Si riscrive ora l'algoritmo di Euclide per calcolare informazioni utili aggiuntive. In particolare, si estende l'algoritmo per calcolare i coefficienti interi x e y tali che

$$d = \text{MCD}(a, b) = ax + by . \tag{33.18}$$

Si noti che x e y possono essere 0 o negativi. In seguito questi coefficienti saranno utili per il calcolo del reciproco modulare. La procedura EXTENDED-EUCLID prende come input una coppia qualsiasi di interi non negativi e restituisce una tripla della forma (d,x,y) che soddisfa l'equazione (33.18).

```
EXTENDED-EUCLID(a,b)
1  if b = 0
2   then return (a,1,0)
3  (d', x', y') ← EXTENDED-EUCLID(b, a mod b)
4  (d, x, y) ← (d', y', x' − ⌊a/b⌋y')
5  return (d, x, y)
```

La figura 33.1 illustra l'esecuzione di EXTENDED-EUCLID con il calcolo di MCD(99,78).

La procedura EXTENDED-EUCLID è una variazione della procedura EUCLID. La linea 1 è equivalente al controllo "$b = 0$" nella linea 1 di EUCLID. Se $b = 0$, allora EXTENDED-EUCLID restituisce non solo $d = a$ nella linea 2, ma anche i coefficienti $x = 1$, $y = 0$, così che $a = ax + by$. Se $b \neq 0$, EXTENDED-EUCLID prima calcola (d', x', y') così che $d' = \text{MCD}(b, a \bmod b)$ e

$$d' = bx' + (a \bmod b)y' . \tag{33.19}$$

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 esempio di funzionamento di* EXTENDED-EUCLID *sugli input* 99 *e* 78. *Ogni riga mostra un livello di ricorsione: i valori degli input a e b, il valore calcolato* $\lfloor a/b \rfloor$ *e i valori d, x e y restituiti. La tripla* (d,x,y) *restituita diventa la tripla* (d',x',y') *che sarà usata nel calcolo al più alto livello di ricorsione successivo. La chiamata* EXTENDED-EUCLID(99,78) restituisce (3,-11,14), *così* MCD(99,78) = 3 *e* MCD(99,78) = 3 = 99·(-11) + 78·14.

Come per EUCLID, si ha in questo caso $d = \text{MCD}(a, b) = d' = \text{MCD}(b, a \bmod b)$. Per ottenere x e y tali che $d = ax + by$, si comincia riscrivendo l'equazione 33.19 usando l'equazione $d = d'$ e l'equazione (33.2):

$$\begin{aligned} d &= bx' + (a - \lfloor a/b \rfloor b)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y'). \end{aligned}$$

Pertanto scegliendo $x = y'$ e $y = x' - \lfloor a/b \rfloor$ si soddisfa l'equazione $d = ax + by$, dimostrando la correttezza di EXTENDED-EUCLID.

Poiché il numero di chiamate ricorsive fatte in EUCLID è uguale al numero di chiamate ricorsive fatte in EXTENDED-EUCLID, il tempo di esecuzione di EUCLID e di EXTENDED-EUCLID è lo stesso, a meno di un fattore costante. Cioè, per $a > b > 0$, il numero di chiamate ricorsive è $O(\lg b)$.

Esercizi

33.2-1 Si dimostri che le equazioni (33.13) e (33.14) implicano l'equazione (33.15).

33.2-2 Si calcolino i valori (d, x, y) restituiti in output della chiamata EXTENDED-EUCLID(899,493).

33.2-3 Si dimostri che per tutti gli interi a, k e n, vale

$$\text{MCD}(a, n) = \text{MCD}(a + kn, n).$$

33.2-4 Si riscriva l'algoritmo EUCLID in forma iterativa in modo che usi solo una quantità costante di memoria (cioè, memorizzi solo un numero costante di valori interi).

33.2-5 Si mostri che, se $a > b \geq 0$, la chiamata EUCLID(a, b) fa al più $1 + \log_\phi b$ chiamate ricorsive. Si migliori questo limite abbassandolo a $1 + \log_\phi(b/\text{MCD}(a, b))$.

33.2-6 Cosa restituisce EXTENDED-EUCLID(F_{k+1}, F_k)? Si dimostri la correttezza della risposta.

33.2-7 Si verifichi l'output (d,x,y) di EXTENDED-EUCLID(a,b) mostrando che se $d|a$, $d|b$ e $d = ax + by$, allora $d = \text{MCD}(a,b)$.

33.2-8 Si definisca la funzione MCD per più di due argomenti con l'equazione ricorsiva $\text{MCD}(a_0, a_1, \ldots, a_n) = \text{MCD}(a_0, \text{MCD}(a_1, \ldots, a_n))$. Si mostri che MCD restituisce le stesse soluzioni indipendentemente dall'ordine con cui sono specificati i suoi argomenti. Si mostri come trovare $x_0, x_1, \ldots, x_n$ tali che $\text{MCD}(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \ldots + a_nx_n$. Si mostri che il numero di divisioni eseguite dall'algoritmo è $O(n + \lg(\max_i a_i))$.

33.2-9 Si definisca $\text{mcm}(a_1, a_2, \ldots, a_n)$ come il ***minimo comune multiplo*** degli interi $a_1, a_2, \ldots, a_n$, cioè, il più piccolo intero non negativo che è multiplo di ogni a_i. Si mostri come calcolare $\text{mcm}(a_1, a_2, \ldots, a_n)$ in modo efficiente usando l'operazione MCD (di due argomenti) come sottoprogramma.

33.2-10 Si dimostri che n_1, n_2, n_3 ed n_4 sono primi tra loro se e solo se $\text{MCD}(n_1n_2, n_3n_4) = \text{MCD}(n_1n_3, n_2n_4) = 1$. Si mostri più in generale che $n_1, n_2, \ldots, n_k$ sono primi tra loro se e solo se gli elementi delle coppie di un insieme di $\lceil \lg k \rceil$ coppie di numeri ottenuti dagli n_i sono primi tra loro.

33.3 Aritmetica modulare

Informalmente, si può pensare all'aritmetica modulare come all'usuale aritmetica dei numeri interi tranne che, operando modulo di n, ogni risultato x va sostituito con l'elemento di $\{0, 1, \ldots, n-1\}$ che è congruente a x modulo n (cioè, x è sostituito con $x \bmod n$). Questo modello informale è sufficiente se ci si limita alle operazioni di addizione, sottrazione e moltiplicazione. Verrà ora descritto un modello più formale per l'aritmetica modulare utilizzando la teoria dei gruppi.

Gruppi finiti

Un ***gruppo*** $(S, \oplus)$ è un insieme S con un'operazione binaria $\oplus$ definita su S per cui valgono le seguenti proprietà.

1. **Chiusura:** per ogni $a, b \in S$, si ha $a \oplus b \in S$.
2. **Identità:** vi è un elemento $e \in S$ tale che $e \oplus a = e \oplus a = a$ per ogni $a \in S$.
3. **Associatività:** per ogni $a, b, c \in S$, si ha $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
4. **Reciproco o opposto:** per ogni $a \in S$, esiste un unico elemento $b \in S$ tale che $a \oplus b = b \oplus a = e$.

Per esempio si consideri il noto gruppo $(\mathbf{Z}, +)$ degli interi $\mathbf{Z}$ sotto l'operazione di addizione: 0 è l'identità, e l'opposto di a è $-a$. Se un gruppo $(S, \oplus)$ soddisfa la ***legge commutativa*** $a \oplus b = b \oplus a$ per ogni $a, b \in S$, allora è un ***gruppo abeliano***. Se un gruppo $(S, \oplus)$ soddisfa $|S| < \infty$, allora è un ***gruppo finito***.

$+_6$	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	4	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

Figura 33.2 *Due gruppi finiti. Le classi di equivalenza sono denotate dai loro elementi rappresentativi.* (a) *Il gruppo* $(\mathbf{Z}_6, +_6)$. (b) *Il gruppo* $(\mathbf{Z}_{15}^*, \cdot_{15})$.

I gruppi definiti dall'addizione e dalla moltiplicazione modulari

Si possono formare due gruppi abeliani finiti usando l'addizione e la moltiplicazione modulo n, dove n è un intero positivo. Questi gruppi sono basati sulle classi di equivalenza degli interi modulo n, definite al paragrafo 33.1.

Per definire un gruppo su $\mathbf{Z}_n$, vi è bisogno di opportune operazioni binarie che si ottengono ridefinendo le comuni operazioni di addizione e moltiplicazione. È facile definire le operazioni di addizione e moltiplicazione su $\mathbf{Z}_n$ perché le classi di equivalenza di due interi determinano univocamente la classe di equivalenza della loro somma o del loro prodotto. Cioè, se $a \equiv a' \pmod{n}$ e $b \equiv b' \pmod{n}$, allora

$$a + b \equiv a' + b' \pmod{n},$$
$$ab \equiv a'b' \pmod{n}.$$

Pertanto, si definiscono l'addizione e la moltiplicazione modulo n, denotate con $+_n$ e $\cdot_n$, come segue:

$$[a]_n +_n [b]_n = [a + b]_n,$$
$$[a]_n \cdot_n [b]_n = [ab]_n.$$

(La sottrazione può essere analogamente definita su $\mathbf{Z}_n$ con $[a]_n -_n [b]_n = [a - b]_n$, ma la divisione è più complicata, come si vedrà). Questi fatti giustificano la pratica comune e conveniente di usare gli elementi non negativi minimi come rappresentanti di ogni classe di equivalenza quando si eseguono calcoli in $\mathbf{Z}_n$. La somma, la sottrazione e la moltiplicazione sono eseguite di solito sui rappresentanti, ma ogni risultato x viene sostituito dal rappresentante della sua classe (cioè da $x \bmod n$).

Usando questa definizione dell'addizione modulo n, si definisce il ***gruppo additivo modulo n*** come $(\mathbf{Z}_n, +_n)$. La cardinalità del gruppo additivo modulo n è $|\mathbf{Z}_n| = n$. La figura 33.2(a) fornisce la tabella dell'operazione del gruppo $(\mathbf{Z}_6, +_6)$.

Teorema 33.12

Il sistema $(\mathbf{Z}_n, +_n)$ è un gruppo abeliano finito.

Dimostrazione. L'associatività e la commutatività di $+_n$ seguono dall'associatività e dalla commutatività di $+$:

$$\begin{aligned}([a]_n +_n [b]_n) +_n [c]_n &= [(a+b)+c]_n \\ &= [a+(b+c)]_n \\ &= [a]_n +_n ([b]_n +_n [c]_n)\,,\end{aligned}$$

$$\begin{aligned}[a]_n +_n [b]_n &= [a+b]_n \\ &= [b+a]_n \\ &= [b]_n +_n [a]_n\,.\end{aligned}$$

L'elemento identità di $(\mathbf{Z}_n, +_n)$ è 0 (cioè $[0]_n$). L'opposto di un elemento a (cioè $[a]_n$) è l'elemento $-a$ (cioè, $[-a]_n$ o $[n-a]_n$), poiché $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$. ■

Usando la definizione di moltiplicazione modulo n, si definisce il ***gruppo moltiplicativo modulo n*** come $(\mathbf{Z}_n^*, \cdot_n)$. Gli elementi di questo gruppo sono l'insieme $\mathbf{Z}_n^*$ degli elementi di $\mathbf{Z}_n$ che sono primi rispetto a n:

$$\mathbf{Z}_n^* = \{[a]_n \in \mathbf{Z}_n : \mathrm{MCD}(a, n) = 1\}\,.$$

Per vedere che $\mathbf{Z}_n^*$ è ben definito, si noti che, per $0 \le a < n$, si ha $a \equiv (a + kn) \pmod n$ per tutti gli interi k. Dall'Esercizio 33.2-3, allora, $\mathrm{MCD}(a, n) = 1$ implica $\mathrm{MCD}(a + kn, n) = 1$ per tutti gli interi k. Poiché $[a]_n = \{a + kn : k \in \mathbf{Z}\}$, l'insieme $\mathbf{Z}_n^*$ è ben definito. Un esempio di tale gruppo è

$$\mathbf{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}\,,$$

dove l'operazione del gruppo è la moltiplicazione modulo 15. (In questo caso si denota un elemento $[a]_{15}$ con a.) La figura 33.2(b) mostra il gruppo $(\mathbf{Z}_{15}^*, \cdot_{15})$. Per esempio, $8 \cdot 11 = 13 \pmod{15}$, operando in $\mathbf{Z}_{15}^*$. L'identità per questo gruppo è 1.

Teorema 33.13

Il sistema $(\mathbf{Z}_n^*, \cdot_n)$ è un gruppo abeliano finito.

Dimostrazione. Il Teorema 33.6 implica che $(\mathbf{Z}_n^*, \cdot_n)$ è chiuso. L'associatività e la commutatività possono essere dimostrate per $\cdot_n$ come si è già fatto per $+_n$ nella dimostrazione del Teorema 33.12. L'elemento identità è $[1]_n$. Per mostrare l'esistenza dei reciproci, sia a un elemento di $\mathbf{Z}_n^*$ e sia (d, x, y) l'output di EXTENDED-EUCLID(a,n). Allora $d = 1$, poiché $a \in \mathbf{Z}_n^*$ e

$$ax + ny = 1$$

oppure, equivalentemente,

$$ax \equiv 1 \pmod n\,.$$

Pertanto, $[x]_n$ è un reciproco di $[a]_n$ modulo n. La dimostrazione che i reciproci sono definiti univocamente è rimandato al corollario 33.26. ■

Nel resto del capitolo, quando si parlerà dei gruppi $(\mathbf{Z}_n, +_n)$ e $(\mathbf{Z}_n^*, \cdot_n)$, si seguirà la comoda pratica di denotare classi di equivalenza con i loro elementi rappresentativi e le operazioni $+_n$ e $\cdot_n$ con le usuali notazioni aritmetiche $+$ e $\cdot$ (o la giustapposizione), rispettivamente. Inoltre, le equivalenze modulo n possono essere interpretate come equazioni in $\mathbf{Z}_n$. Per esempio, le due seguenti espressioni sono equivalenti:

$$\begin{aligned}ax &\equiv b \pmod n\,,\\ [a]_n \cdot_n [x]_n &= [b]_n\,.\end{aligned}$$

Inoltre, ancora per convenienza, talvolta ci si riferirà a un gruppo $(S, \oplus)$ semplicemente con S quando l'operazione sia chiara dal contesto. Quindi si farà riferimento a $(\mathbf{Z}_n, +_n)$ e $(\mathbf{Z}_n^*, \cdot_n)$ come $\mathbf{Z}_n$ e $\mathbf{Z}_n^*$, rispettivamente.

Il reciproco di un elemento a è denotato con $(a^{-1} \bmod n)$. La divisione in $\mathbf{Z}_n^*$ è definita dall'equazione $a/b \equiv ab^{-1} \pmod n$. Per esempio in $\mathbf{Z}_{15}^*$ si ha $7^{-1} \equiv 13 \pmod{15}$, poiché $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$, così che $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

La cardinalità di $\mathbf{Z}_n^*$ è denotata con $\phi(n)$. Questa funzione, conosciuta come la ***funzione fi di Eulero***, soddisfa l'equazione

$$\phi(n) = n \prod_{p \mid n} \left(1 - \frac{1}{p}\right)\,, \tag{33.20}$$

dove p varia su tutti i primi che dividono n (compreso n se n è primo). Si dimostrerà ora questa formula. Intuitivamente, si comincia con una lista degli n resti $\{0, 1, \ldots, n-1\}$ e poi, per ogni primo p che divide n, si cancella ogni multiplo di p nella lista. Per esempio, poiché i divisori primi di 45 sono 3 e 5,

$$\begin{aligned}\phi(45) &= 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right)\\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right)\\ &= 24.\end{aligned}$$

Se p è primo, allora $\mathbf{Z}_p^* = \{1, 2, \ldots, p-1\}$ e

$$\phi(p) = p - 1\,. \tag{33.21}$$

Se n è composto, allora $\phi(n) < n - 1$.

Sottogruppi

Se $(S, \oplus)$ è un gruppo, $S' \subseteq S$ e anche $(S', \oplus)$ è un gruppo, allora $(S', \oplus)$ è detto ***sottogruppo*** di $(S, \oplus)$. Per esempio, gli interi pari formano un sottogruppo degli interi rispetto all'operazione di addizione. Il seguente teorema fornisce uno strumento utile per il riconoscimento dei sottogruppi.

Teorema 33.14 (Un sottoinsieme chiuso di un gruppo finito è un sottogruppo)

Se $(S, \oplus)$ è un gruppo finito e S' è un qualsiasi sottoinsieme non vuoto di S tale che $a \oplus b \in S'$ per ogni $a, b \in S'$, allora $(S', \oplus)$ è un sottogruppo di $(S, \oplus)$.

Dimostrazione. La dimostrazione è lasciata come Esercizio 33.3-2. ■

Per esempio, l'insieme $\{0, 2, 4, 6\}$ forma un sottogruppo di $\mathbf{Z}_8$ poiché è chiuso rispetto all'operazione + (cioè è chiuso rispetto a $+_8$).

Il seguente teorema fornisce un vincolo estremamente utile sulla cardinalità di un sottogruppo.

Teorema 33.15 (Teorema di Lagrange)

Se $(S, \oplus)$ è un gruppo finito e $(S', \oplus)$ è un sottogruppo di $(S, \oplus)$, allora $|S'|$ è un divisore di $|S|$. ■

Un sottogruppo S' di un gruppo S è detto sottogruppo *proprio* se $S' \neq S$. Il seguente corollario sarà usato al paragrafo 33.8 nell'analisi della procedura di Miller-Rabin per la verifica di primalità.

Corollario 33.16

Se S' è un sottogruppo proprio di un gruppo finito S, allora $|S'| \leq |S|/2$. ■

Sottogruppi generati da un elemento

Il Teorema 33.14 fornisce un modo interessante per produrre un sottogruppo di un gruppo finito $(S, \oplus)$: si sceglie un elemento a e si prendono tutti gli elementi che possono essere generati da a usando l'operazione del gruppo. In particolare, si definisce $a^{(k)}$ per $k \geq 1$ con

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \ldots \oplus a}_{k \text{ volte}} .$$

Per esempio, se si prende $a = 2$ nel gruppo $\mathbf{Z}_6$, la sequenza $a^{(1)}, a^{(2)}, \ldots$ è $2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots$. Nel gruppo $\mathbf{Z}_n$ si ha $a^{(k)} = ka \bmod n$ e nel gruppo $\mathbf{Z}_n^*$ si ha $a^{(k)} = a^k \bmod n$. Il ***sottogruppo generato da a***, denotato con $\langle a \rangle$ oppure con $(\langle a \rangle, \oplus)$, è definito da

$$\langle a \rangle = \{a^{(k)} : k \geq 1\} .$$

Si dice che a ***genera*** il sottogruppo $\langle a \rangle$ o che a è un ***generatore*** di $\langle a \rangle$. Poiché S è finito, $\langle a \rangle$ è un sottoinsieme finito di S, eventualmente comprendente tutto S. Poiché l'associatività di $\oplus$ implica

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)} ,$$

$\langle a \rangle$ è chiuso e quindi, per il Teorema 33.14, $\langle a \rangle$ è un sottogruppo di S. Per esempio, in $\mathbf{Z}_6$, si ha

$$\langle 0 \rangle = \{0\} ,$$
$$\langle 1 \rangle = \{0,1,2,3,4,5\} ,$$
$$\langle 2 \rangle = \{0,2,4\} .$$

Analogamente, in $\mathbf{Z}_7^*$, si ha

$$\langle 1 \rangle = \{1\} ,$$
$$\langle 2 \rangle = \{1,2,4\} ,$$

$$\langle 3 \rangle = \{1,2,3,4,5,6\} .$$

L'*ordine* di a (nel gruppo S), denotato con $\mathrm{ord}(a)$, è definito come il minimo $t > 0$ tale che $a^{(t)} = e$.

Teorema 33.17

Per qualsiasi gruppo $(S, \oplus)$ finito e per ogni $a \in S$, l'ordine di un elemento è uguale alla cardinalità del sottogruppo che genera, cioè, $\mathrm{ord}(a) = |\langle a \rangle|$.

Dimostrazione. Sia $t = \mathrm{ord}(a)$. Poiché $a^{(t)} = e$ e $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ per $k \geq 1$, se $i > t$, allora $a^{(i)} = a^{(j)}$ per qualche $j < i$. Quindi, non ci sono altri elementi dopo $a^{(t)}$ e $\langle a \rangle = \{a^{(1)}, a^{(2)}, \ldots, a^{(t)}\}$. Per dimostrare che $|\langle a \rangle| = t$ si supponga per assurdo che $a^{(i)} = a^{(j)}$ per qualche i e j tali che $1 \leq i < j \leq t$. Allora $a^{(i+k)} = a^{(j+k)}$ per $k \geq 0$. Ma questo implica che $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, una contraddizione, dato che $i + (t - j) < t$ ma t è il valore positivo più piccolo tale che $a^{(t)} = e$. Pertanto, ciascun elemento della sequenza $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ è distinto e $|\langle a \rangle| = t$. ■

Corollario 33.18

La sequenza $a^{(1)}, a^{(2)}, \ldots$ è periodica con periodo $t = \mathrm{ord}(a)$; cioè $a^{(i)} = a^{(j)}$ se e solo se $i \equiv j \pmod t$. ■

Per il corollario precedente si può definire $a^{(0)}$ come e e $a^{(i)}$ come $a^{(i \bmod t)}$ per ogni intero i.

Corollario 33.19

Se $(S, \oplus)$ è un gruppo finito con identità e, allora per ogni $a \in S$ vale $a^{(|S|)} = e$.

Dimostrazione. Il Teorema di Lagrange implica che $\mathrm{ord}(a) \mid |S|$ e così $|S| \equiv 0 \pmod t$, dove $t = \mathrm{ord}(a)$. ■

Esercizi

33.3-1 Si considerino le tabelle delle operazioni per i gruppi $(\mathbf{Z}_4, +_4)$ e $(\mathbf{Z}_5^*, \cdot_5)$. Si mostri che questi sottogruppi sono isomorfi esibendo una corrispondenza biunivoca α tra i loro elementi tale che $a + b \equiv c \pmod 4$ se e solo se $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod 5$.

33.3-2 Si dimostri il Teorema 33.14.

33.3-3 Si mostri che se p è primo ed e è un intero positivo, allora $\phi(p^e) = p^{e-1}(p-1)$.

33.3-4 Si mostri che per qualunque $n > 1$ e per qualunque $a \in \mathbf{Z}_n^*$, la funzione $f_a : \mathbf{Z}_n^* \to \mathbf{Z}_n^*$ definita da $f_a(x) = ax \bmod n$ è una permutazione di $\mathbf{Z}_n^*$.

33.3-5 Si elenchino tutti i sottogruppi di $\mathbf{Z}_9$ e di $\mathbf{Z}_{13}^*$.

33.4 Risoluzione di equazioni lineari modulari

Si considererà ora il problema, rilevante nella pratica, di trovare le soluzioni dell'equazione

$$ax \equiv b \pmod{n}\,, \tag{33.22}$$

dove $n > 0$. Si assume che siano dati a, b ed n e si debbano trovare le x modulo n che soddisfino l'equazione (33.22). Può esserci una, nessuna o più d'una soluzione.

Sia $\langle a \rangle$ il sottogruppo di $\mathbf{Z}_n$ generato da a. Poiché $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, l'equazione (33.22) ha una soluzione se e solo se $b \in \langle a \rangle$. Il Teorema di Lagrange (Teorema 33.15) dice che $|\langle a \rangle|$ deve essere un divisore di n. Il seguente teorema fornisce una precisa caratterizzazione di $\langle a \rangle$.

Teorema 33.20

Per qualsiasi a e n interi positivi, se $d = \mathrm{MCD}(a, n)$, allora

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\}\,, \tag{33.23}$$

e pertanto

$$|\langle a \rangle| = n/d\,.$$

Dimostrazione. Si comincia mostrando che $d \in \langle a \rangle$. Si osservi che EXTENDED-EUCLID(a, n) produce gli interi x' e y' tali che $ax' + ny' = d$. Quindi $ax' \equiv d \pmod{n}$, così che $d \in \langle a \rangle$.

Poiché $d \in \langle a \rangle$, ogni multiplo di d appartiene ad $\langle a \rangle$ perché un multiplo di un multiplo di a è un multiplo di a. Perciò, $\langle a \rangle$ contiene ogni elemento in $\{0, d, 2d, \ldots, ((n/d) - 1)d\}$. Cioè, $\langle d \rangle \subseteq \langle a \rangle$.

Si mostra ora che $\langle a \rangle \subseteq \langle d \rangle$. Se $m \in \langle a \rangle$, allora $m = ax \bmod n$ per qualche intero x e così $m = ax + ny$ per qualche intero y. Però $d|a$ e $d|n$, quindi $d|m$ dall'equazione (33.6). Pertanto $m \in \langle d \rangle$.

Combinando questi risultati, si ha che $\langle a \rangle = \langle d \rangle$. Per vedere che $|\langle a \rangle| = n/d$, si osservi che vi sono esattamente n/d multipli di d tra 0 ed $n - 1$ inclusi. ■

Corollario 33.21

L'equazione $ax \equiv b \pmod{n}$ è risolubile nell'incognita x se e solo se $\mathrm{MCD}(a, n)|b$. ■

Corollario 33.22

L'equazione $ax \equiv b \pmod{n}$ ha d soluzioni distinte modulo n, dove $d = \mathrm{MCD}(a, n)$, oppure non ha soluzioni.

Dimostrazione. Se $ax \equiv b \pmod{n}$ ha una soluzione, allora $b \in \langle a \rangle$. La sequenza $ai \bmod n$, per $i = 0, 1, \ldots$ è periodica con periodo $|\langle a \rangle| = n/d$, per il Corollario 33.18. Se $b \in \langle a \rangle$, allora b appare esattamente d volte nella sequenza $ai \bmod n$, per $i = 0, 1, \ldots, n - 1$, poiché il blocco di valori $\langle a \rangle$ di lunghezza (n/d) è ripetuto esattamente d volte con i crescente da 0 a $n - 1$. Gli indici x di queste d posizioni sono le soluzioni dell'equazione $ax \equiv b \pmod{n}$. ■

Teorema 33.23

Sia $d = \mathrm{MCD}(a, n)$ e si supponga che $d = ax' + ny'$, per qualche intero x' e y' (per esempio come quelli calcolati nella procedura EXTENDED-EUCLID). Se $d|b$, allora una delle soluzioni dell'equazione $ax \equiv b \pmod{n}$ è il valore x_0, dove

$$x_0 = x'(b/d) \bmod n\,.$$

Dimostrazione. Poiché $ax' \equiv b \pmod{n}$, si ha

$$\begin{aligned} ax_0 &\equiv ax'(b/d) && \pmod{n} \\ &\equiv d'(b/d) && \pmod{n} \\ &\equiv b && \pmod{n} \end{aligned}$$

e così x_0 è una soluzione di $ax \equiv b \pmod{n}$. ■

Teorema 33.24

Si supponga che l'equazione $ax \equiv b \pmod{n}$ sia risolubile (cioè $d|b$, dove $d = \mathrm{MCD}(a, n)$) e che x_0 sia una soluzione. Allora quest'equazione ha esattamente d soluzioni distinte modulo n date da $x_i = x_0 + i(n/d)$, per $i = 1, 2, \ldots, d - 1$.

Dimostrazione. Poiché $n/d > 0$ e $0 \le i(n/d) < n$, per $i = 0, 1, \ldots, d - 1$, i valori $x_0, x_1, \ldots, x_{d-1}$ sono tutti distinti, modulo n. Dalla periodicità della sequenza $ai \bmod n$ (corollario 33.18), se x_0 è una soluzione di $ax \equiv b \pmod{n}$, allora ogni x_i è una soluzione. Per il corollario 33.22, vi sono esattamente d soluzioni, quindi $x_0, x_1, \ldots, x_{d-1}$ sono tutte le soluzioni. ■

Si sono così presentati tutti gli strumenti matematici necessari a risolvere l'equazione $ax \equiv b \pmod{n}$; il seguente algoritmo stampa tutte le soluzioni di quest'equazione. Gli input a e b sono interi arbitrari e n è un intero positivo arbitrario.

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') ← EXTENDED-EUCLID(a, n)
2  if d|b
3     then  x0 ← x'(b/d) mod n
4        for i ← 0 to d − 1
5           do stampa (x0 + i(n/d)) mod n
6     else stampa "nessuna soluzione"
```

Come esempio di funzionamento di questa procedura, si consideri l'equazione $14x \equiv 30 \pmod{100}$ (in questo caso, $a = 14$, $b = 30$ e $n = 100$). Chiamando EXTENDED-EUCLID alla linea 1, si ottiene $(d, x, y) = (2, -7, 1)$. Poiché $2|30$, sono eseguite le linee 3-5. Alla linea 3, si calcola $x_0 = (-7)(15) \bmod 100 = 95$. Il ciclo alle linee 4-5 stampa le due soluzioni: 95 e 45.

La procedura MODULAR-LINEAR-EQUATION-SOLVER funziona come segue. La linea 1 calcola sia $d = \mathrm{MCD}(a, n)$ che i due valori x' e y' tali che $d = ax' + ny'$, dimostrando che x' è una soluzione di $ax' \equiv d \pmod{n}$. Se d non divide b, allora l'equazione $ax \equiv b \pmod{n}$ non ha soluzioni, per il corollario 33.21. La linea 2 controlla se d non divide b, nel qual caso la linea 6 riporta che non vi sono soluzioni. Altrimenti alla linea 3 si calcola una soluzione x_0 dell'equazione (33.22), secondo il Teorema 33.23. Data una soluzione, il Teorema 33.24 stabilisce che le altre $d - 1$ soluzioni possono essere ottenute aggiungendo multipli di (n/d), modulo n. Il ciclo **for** alle linee 4-5 stampa tutte le d soluzioni, cominciando con x_0 e saltando di (n/d) in (n/d) modulo n.

MODULAR-LINEAR-EQUATION-SOLVER richiede tempo di esecuzione $O(\lg n + \text{MCD}(a, n))$ per le operazioni aritmetiche, poiché EXTENDED-EUCLID richiede $O(\lg n)$ operazioni aritmetiche ed ogni iterazione del ciclo **for** alle linee 4-5 richiede un numero costante di operazioni aritmetiche.

I seguenti corollari del Teorema 33.24 forniscono caratterizzazioni di particolare interesse.

Corollario 33.25

Per qualsiasi $n > 1$, se $\text{MCD}(a, n) = 1$, allora l'equazione $ax \equiv b \pmod{n}$ ha un'unica soluzione modulo n. ■

Se $b = 1$, un caso comune di notevole interesse, la soluzione x che si sta cercando è il ***reciproco*** di a, modulo n.

Corollario 33.26

Per qualsiasi $n > 1$, se $\text{MCD}(a, n) = 1$, allora l'equazione

$$ax \equiv 1 \pmod{n} \tag{33.24}$$

ha un'unica soluzione, modulo n. Altrimenti non ha soluzioni. ■

Il corollario 33.26 permette di usare la notazione $(a^{-1} \bmod n)$ per fare riferimento al reciproco di a modulo n, quando a e n sono primi tra loro. Se $\text{MCD}(a, n) = 1$, allora una soluzione dell'equazione $ax \equiv 1 \pmod{n}$ è l'intero x restituito da EXTENDED-EUCLID, poiché l'equazione

$$\text{MCD}(a, n) = 1 = ax + ny$$

implica $ax \equiv 1 \pmod{n}$. Pertanto, $(a^{-1} \bmod n)$ può essere calcolata efficientemente con EXTENDED-EUCLID.

Esercizi

33.4-1 Si trovino tutte le soluzioni dell'equazione $35x \equiv 10 \pmod{50}$.

33.4-2 Si dimostri che l'equazione $ax \equiv ay \pmod{n}$ implica $x \equiv y \pmod{n}$ per $\text{MCD}(a, n) = 1$. Si mostri che la condizione $\text{MCD}(a, n) = 1$ è necessaria fornendo un controesempio con $\text{MCD}(a, n) > 1$.

33.4-3 Si consideri il seguente cambiamento della linea 3 di MODULAR-LINEAR-EQUATION-SOLVER:

3 **then** $x_0 \leftarrow x'(b/d) \bmod (n/d)$.

Funziona? Giustificare la risposta.

*** 33.4-4** Sia $f(x) \equiv f_0 + f_1 x + \ldots + f_t x^t \pmod{p}$ un polinomio di grado t con coefficienti f_i presi da $\mathbf{Z}_p$, dove p è primo. Si dice che $a \in \mathbf{Z}_p$ è uno *zero* di f se $f(a) \equiv 0 \pmod{p}$. Si dimostri che se a è uno zero di f, allora $f(x) \equiv (x - a)g(x) \pmod{p}$ per qualche polinomio $g(x)$ di grado $t - 1$. Si provi per induzione su t che un polinomio $f(x)$ di grado t può avere al più t zeri distinti modulo p dove p è primo.

33.5 Il teorema cinese del resto

All'incirca nel 100 d.C., il matematico cinese Sun-Tsù risolse il problema di trovare gli interi x che danno come resto 2, 3 e 2 quando vengono divisi per 3, 5 e 7, rispettivamente. Una delle soluzione è $x = 23$; tutte le soluzioni sono della forma $23 + 105k$ per k intero arbitrario. Il "teorema cinese del resto" fornisce una corrispondenza tra un sistema di equazioni modulo un insieme di moduli primi fra loro a coppie (per esempio, 3, 5 e 7) e un'equazione modulo il loro prodotto (per esempio, 105).

Il teorema cinese del resto ha due usi principali. Sia l'intero $n = n_1 n_2 \ldots n_k$, dove i fattori n_i sono primi tra loro. In primo luogo, il teorema cinese del resto è un "teorema di struttura" descrittivo che descrive la struttura di $\mathbf{Z}_n$ come identica a quella del prodotto cartesiano $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \ldots \times \mathbf{Z}_{n_k}$ con l'addizione componente per componente e la moltiplicazione modulo n_i nella i-esima componente. In secondo luogo, questa descrizione può essere spesso usata per ottenere algoritmi efficienti poiché operare in ognuno dei sistemi $\mathbf{Z}_{n_i}$ può rivelarsi più efficiente (in termini di operazioni sui bit) che lavorare modulo n.

Teorema 33.27 (Teorema cinese del resto)

Sia $n = n_1 n_2 \ldots n_k$, dove i fattori n_i sono primi tra loro. Si consideri la corrispondenza

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) , \tag{33.25}$$

dove $a \in \mathbf{Z}_n$, $a_i \in \mathbf{Z}_{n_i}$ e

$$a_i = a \bmod n_i$$

per $i = 1, 2, \ldots, k$. Allora, la corrispondenza (33.25) tra $\mathbf{Z}_n$ e il prodotto cartesiano $\mathbf{Z}_{n_1} \times \mathbf{Z}_{n_2} \times \ldots \times \mathbf{Z}_{n_k}$ è biunivoca. Le operazioni eseguite sugli elementi di $\mathbf{Z}_n$ possono essere eseguite in modo equivalente sulle corrispondenti k-uple eseguendo le operazioni indipendentemente per ogni coordinata nel sistema appropriato. Cioè, se

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) ,$$

$$b \leftrightarrow (b_1, b_2, \ldots, b_k) ,$$

allora

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k) , \tag{33.26}$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k) , \tag{33.27}$$

$$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \ldots, a_k b_k \bmod n_k) . \tag{33.28}$$

Dimostrazione. La trasformazione tra le due rappresentazioni è abbastanza semplice. Per passare da a a $(a_1, a_2, \ldots, a_k)$ sono richieste solo k divisioni. Calcolare a partendo dalle $(a_1, a_2, \ldots, a_k)$ è altrettanto semplice utilizzando la seguente formula. Sia $m_i = n/n_i$, per $i = 1, 2, \ldots, k$. Si noti che $m_i = n_1 n_2 \ldots n_{i-1} n_{i+1} \ldots n_k$ così che $m_i \equiv 0 \pmod{n_j}$ per ogni $j \neq i$. Allora, ponendo

$$c_i = m_i(m_i^{-1} \bmod n_i), \tag{33.29}$$

per $i = 1, 2, \ldots, k$, si ha

$$a \equiv (a_1c_1 + a_2c_2 + \ldots + a_kc_k) \pmod{n}. \tag{33.30}$$

L'equazione 33.29 è ben definita, poiché m_i e n_i sono primi tra loro (per il Teorema 33.6), così il corollario 33.26 implica che $(m_i^{-1} \bmod n_i)$ sia definito. Per verificare l'equazione (33.30), si noti che $c_j \equiv m_j \equiv 0 \pmod{n_i}$. Pertanto si ha la corrispondenza

$$c_i \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0),$$

un vettore che ha 0 dappertutto tranne che nella i-esima coordinata dove ha un 1. Le c_i formano così, in un certo senso, una "base" per la rappresentazione. Per ogni i allora si ha

$$\begin{aligned} a &\equiv a_ic_i && \pmod{n_i} \\ &\equiv a_im_i(m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &\equiv a_i && \pmod{n_i}. \end{aligned}$$

Poiché si può eseguire la trasformazione in entrambi i versi, la corrispondenza è biunivoca. Le equazioni (33.26)-(33.28) seguono direttamente dall'Esercizio 33.1-6, poiché $x \bmod n_i = (x \bmod n) \bmod n_i$ per qualsiasi x e $i = 1, 2, \ldots, k$. ■

I seguenti corollari saranno usati successivamente nel capitolo.

Corollario 33.28

Se $n_1, n_2, \ldots, n_k$ sono primi tra loro a coppie ed $n = n_1n_2 \ldots n_k$, allora per k interi $a_1, a_2, \ldots, a_k$ qualsiasi, il sistema di equazioni

$$x \equiv a_i \pmod{n_i}$$

per $i = 1, 2, \ldots, k$, ha un'unica soluzione modulo n nell'incognita x. ■

Corollario 33.29

Se $n_1, n_2, \ldots, n_k$ sono primi tra loro a coppie e $n = n_1n_2 \ldots n_k$ allora, per tutti gli interi a e x,

$$x \equiv a \pmod{n_i}$$

per $i = 1, 2, \ldots, k$ se e solo se

$$x \equiv a \pmod{n}.$$

■

Come esempio del teorema cinese del resto, si supponga di avere le due equazioni

$$a \equiv 2 \pmod{5},$$
$$a \equiv 3 \pmod{13},$$

per cui $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$, $n_2 = m_1 = 13$ e si voglia calcolare $a \bmod 65$, essendo $n = 65$. Dato che $13^{-1} \equiv 2 \pmod 5$ e $5^{-1} \equiv 8 \pmod{13}$, si ha

$$c_1 = 13(2 \bmod 5) = 26,$$
$$c_2 = 5(8 \bmod 13) = 40.$$

	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 *Un'illustrazione del teorema del resto cinese per $n_1 = 5$ e $n_2 = 13$. In questo esempio, $c_1 = 26$ e $c_2 = 40$. Nell'intersezione tra la riga i e la colonna j è mostrato il valore di a modulo 65 tale che $(a \bmod 5) = i$ e $(a \bmod 65) = j$. Si noti che l'intersezione tra la riga 0 e la colonna 0 contiene 0. Analogamente l'intersezione tra la riga 4 e la colonna 12 contiene 64 (congruente a -1). Poiché $c_1 = 26$, spostandosi in basso di una riga, si incrementa a di 26. Analogamente, $c_2 = 40$ significa che muovendosi a destra di una colonna si incrementa a di 40. Aumentare a di 1 corrisponde a spostarsi diagonalmente verso il basso e a destra, richiudendo circolarmente le diagonali dal basso all'alto e da destra a sinistra.*

e

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 && \pmod{65} \\ &\equiv 52 + 120 && \pmod{65} \\ &\equiv 42 && \pmod{65}. \end{aligned}$$

Si veda la figura 33.3 per un'illustrazione del teorema cinese del resto modulo 65.

Quindi per operare modulo n si può usare direttamente il modulo n, oppure usare la rappresentazione trasformata che impiega i calcoli modulo n_i separatamente, a seconda di come risulta più conveniente. I calcoli sono assolutamente equivalenti.

Esercizi

33.5-1 Si trovino tutte le soluzioni delle equazioni $x \equiv 4 \pmod 5$ e $x \equiv 5 \pmod{11}$.

33.5-2 Si trovino tutti gli interi x che, divisi per 2, 3, 4, 5 e 6, danno come resto 1, 2, 3, 4, 5, rispettivamente.

33.5-3 Si spieghi che, in base alle definizioni del Teorema 33.27, se $\mathrm{MCD}(a,n) = 1$, allora

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_k)).$$

33.5-4 In base alle definizioni del Teorema 33.27, si dimostri che il numero di radici dell'equazione $f(x) \equiv 0 \pmod n$ è uguale al prodotto del numero di radici di ogni equazione $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}$, ..., $f(x) \equiv 0 \pmod{n_k}$.

33.6 Potenze di un elemento

Proprio come è naturale considerare i multipli di un dato elemento a modulo n, spesso è naturale considerare la sequenza delle potenze di a modulo n, dove $a \in \mathbb{Z}_n^*$:

$$a^0, a^1, a^2, a^3, \ldots, \tag{33.31}$$

modulo n. Indicizzando da 0, il valore 0-esimo di questa sequenza è $a^0 \bmod n = 1$ e l'i-esimo valore è $a^i \bmod n$. Per esempio, le potenze di 3 modulo 7 sono

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	...

mentre le potenze di 2 modulo 7 sono

i	0	1	2	3	4	5	6	7	8	9	10	11	...
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	...

In questo paragrafo, con $\langle a \rangle$ si denota il sottogruppo di $\mathbf{Z}_n^*$ generato da a e con $\mathrm{ord}_n(a)$ ("l'ordine di a modulo n") si denota l'ordine di a in $\mathbf{Z}_n^*$. Per esempio, $\langle 2 \rangle = \{1,2,4\}$ in $\mathbf{Z}_7^*$ e $\mathrm{ord}_7(2) = 3$. Usando la definizione della funzione $\phi(n)$ di Eulero come dimensione di $\mathbf{Z}_n^*$ (si veda il paragrafo 33.3), si può enunciare il Corollario 33.19 nella notazione di $\mathbf{Z}_n^*$ per ottenere il Teorema di Eulero e la sua specializzazione per $\mathbf{Z}_p^*$, dove p è primo, per ottenere anche il Teorema di Fermat.

Teorema 33.30 (Teorema di Eulero)

Per qualsiasi intero $n > 1$,

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ per ogni } a \in \mathbf{Z}_n^*. \tag{33.32}$$

Teorema 33.31 (Teorema di Fermat)

Se p è primo, allora

$$a^{p-1} \equiv 1 \pmod{p} \text{ per ogni } a \in \mathbf{Z}_p^*. \tag{33.33}$$

Dimostrazione. Per l'equazione (33.21), $\phi(p) = p - 1$ se p è primo. ■

Questo corollario è verificato da ogni elemento di $\mathbf{Z}_p$ tranne lo 0, dato che $0 \notin \mathbf{Z}_p^*$. Tuttavia, per ogni $a \in \mathbf{Z}_p$, si ha che $a^p \equiv a \pmod{p}$ se p è primo.

Se $\mathrm{ord}_n(g) = |\mathbf{Z}_n^*|$, allora ogni elemento in $\mathbf{Z}_n^*$ è una potenza di g modulo n e si dice che g è una ***radice primitiva*** oppure un ***generatore*** di $\mathbf{Z}_n^*$. Per esempio, 3 è una radice primitiva, modulo 7. Se $\mathbf{Z}_n^*$ ha una radice primitiva, si dice che il gruppo $\mathbf{Z}_n^*$ è ***ciclico***. Si omette la dimostrazione del seguente teorema, che è provato da Niven e Zuckerman [151].

Teorema 33.32

I valori di $n > 1$ per cui $\mathbf{Z}_n^*$ è ciclico sono 2, 4, p^e e $2p^e$, per tutti i numeri primi dispari p e per tutti gli interi positivi e.

Se g è una radice primitiva di $\mathbf{Z}_n^*$ ed a un qualsiasi elemento di $\mathbf{Z}_n^*$, allora esiste un valore z tale che $g^z \equiv a \pmod{n}$. Questo z è chiamato il ***logaritmo discreto*** oppure l'***indice*** di a modulo n in base g; questo valore si denota con $\mathrm{ind}_{n,g}(a)$.

Teorema 33.33 (Teorema del logaritmo discreto)

Se g è una radice primitiva di $\mathbf{Z}_n^*$ allora l'equazione $g^x \equiv g^y \pmod{n}$ vale se e solo se vale l'equazione $x \equiv y \pmod{\phi(n)}$.

Dimostrazione. Si supponga prima che $x \equiv y \pmod{\phi(n)}$. Allora, $x = y + k\,\phi(n)$ per qualche intero k. Quindi,

$$\begin{aligned} g^x &\equiv g^{y + k\phi(n)} && \pmod{n} \\ &\equiv g^y \cdot (g^{\phi(n)})^k && \pmod{n} \\ &\equiv g^y \cdot 1^k && \pmod{n} \\ &\equiv g^y && \pmod{n}. \end{aligned}$$

Viceversa, si supponga che $g^x \equiv g^y \pmod{n}$. Poiché la sequenza di potenze di g genera ogni elemento di $\langle g \rangle$ e $|\langle g \rangle| = \phi(n)$, il corollario 33.18 implica che la sequenza di potenze di g è periodica con periodo $\phi(n)$. Quindi, se $g^x \equiv g^y \pmod{n}$, allora deve essere $x \equiv y \pmod{\phi(n)}$. ■

Considerare logaritmi discreti può talvolta semplificare il ragionamento sulle equazioni modulari, come è mostrato nella dimostrazione del seguente teorema.

Teorema 33.34

Se p è un primo dispari ed $e \geq 1$, allora l'equazione

$$x^2 \equiv 1 \pmod{p^e} \tag{33.34}$$

ha solo due soluzioni, $x = 1$ e $x = -1$.

Dimostrazione. Sia $n = p^e$. Il Teorema 33.32 implica che $\mathbf{Z}_n^*$ ha una radice primitiva g. L'equazione (33.34) può essere scritta come

$$(g^{\mathrm{ind}_{n,g}(x)})^2 \equiv g^{\mathrm{ind}_{n,g}(1)} \pmod{n}. \tag{33.35}$$

Dopo aver notato che $\mathrm{ind}_{n,g}(1) = 0$, si osservi che il Teorema 33.33 implica che l'equazione (33.35) sia equivalente a

$$2 \cdot \mathrm{ind}_{n,g}(x) \equiv 0 \pmod{\phi(n)}. \tag{33.36}$$

Per risolvere quest'equazione nell'incognita $\mathrm{ind}_{n,g}(1)$, si applica il metodo del paragrafo 33.4. Ponendo $d = \mathrm{MCD}(2, \phi(n)) = \mathrm{MCD}(2, (p-1)p^{e-1}) = 2$ e notando che $d \mid 0$, si ottiene, per il Teorema 33.24, che l'equazione (33.36) ha esattamente $d = 2$ soluzioni. Allora l'equazione (33.34) ha esattamente 2 soluzioni che sono $x = 1$ e $x = -1$ per verifica diretta. ■

Un numero x è una ***radice quadrata non banale di*** 1 ***modulo*** n se soddisfa l'equazione $x^2 \equiv 1 \pmod{n}$. Per esempio, 6 è una radice quadrata non banale di 1 modulo 35. Il seguente corollario del Teorema 33.34 sarà usato nella prova di correttezza della procedura di verifica di primalità di Miller-Rabin nel paragrafo 33.8.

Corollario 33.35

Se esiste una radice quadrata non banale di 1 modulo n, allora n è composto.

Dimostrazione. Questo corollario è proprio l'antitesi del Teorema 33.34. Se esiste una radice quadrata non banale di 1 modulo n, allora n non può essere primo né potenza di un primo. ■

Elevamento a potenza con quadrature ripetute

Un'operazione che si verifica di frequente nei calcoli della teoria dei numeri è l'elevamento di un numero a potenza modulo un altro numero, operazione conosciuta anche come ***elevamento a potenza modulare***. Più precisamente, si vorrebbe un modo efficiente per calcolare $a^b \bmod n$, dove a e b sono interi non negativi e n è intero positivo. L'elevamento a potenza modulare è anche un'operazione essenziale in molte procedure di verifica di primalità e nel sistema di crittografia a chiave pubblica RSA. Il metodo delle ***quadrature ripetute*** risolve questo problema efficientemente usando la rappresentazione binaria di b. Sia $\langle b_k, b_{k-1}, \ldots, b_1, b_0\rangle$ la rappresentazione binaria di b. (Cioè, la rappresentazione binaria è lunga $k+1$ bit, b_k è il bit più significativo e b_0 è il meno significativo.) La seguente procedura calcola $a^c \bmod n$ aumentando c attraverso raddoppi e incrementi da 0 a b.

```
MODULAR-EXPONENTIATION(a,b,n)
c ← 0
d ← 1
sia ⟨b_k, b_{k-1}, ..., b_1, b_0⟩ la rappresentazione binaria di b
for i ← k downto 0
    do c ← 2c
       d ← (d · d) mod n
       if b_i = 1
          then c ← c + 1
               d ← (d · a) mod n
return d
```

Ogni esponente calcolato in una sequenza è il doppio del precedente esponente oppure il precedente esponente incrementato di 1; la rappresentazione binaria di b è letta da destra a sinistra per controllare quali operazioni sono eseguite. Ogni iterazione del ciclo usa una delle identità

$$a^{2c} \bmod n = (a^c)^2 \bmod n,$$
$$a^{2c+1} \bmod n = a \cdot (a^c)^2 \bmod n,$$

a seconda che $b_i = 0$ o 1, rispettivamente. L'uso essenziale del quadrato ad ogni iterazione spiega il nome "quadrature ripetute". Appena il bit b_i è letto ed elaborato, il valore di c è lo stesso del prefisso $\langle b_k, b_{k-1}, \ldots, b_i\rangle$ della rappresentazione binaria di b. Per esempio, per $a = 7$, $b = 560$ e $n = 561$, l'algoritmo calcola la sequenza dei valori modulo 561 mostrata in figura 33.4; la sequenza di esponenti usati è mostrata alla riga c della tabella.

La variabile c non è in realtà necessaria nell'algoritmo, ma è stata aggiunta per poter dare una spiegazione migliore: l'algoritmo mantiene l'invariante che $d = a^c \bmod n$ al crescere di c con raddoppi e incrementi finché $c = b$. Se gli input a, b e n sono numeri di β bit, allora è richiesto un numero totale $O(\beta^3)$ di operazioni sui bit.

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 *I risultati di* MODULAR-EXPONENTIATION *quando si calcola* $a^b \pmod n$, *con* $a = 7$, $b = 560 = \langle 1000110000\rangle$ *e* $n = 561$. *I valori sono mostrati dopo ogni esecuzione del ciclo* **for**. *Il risultato finale è* 1.

Esercizi

33.6-1 Si disegni una tabella che mostri l'ordine di ogni elemento in $\mathbf{Z}_{11}^*$. Si prenda la radice primitiva g più piccola e si calcoli una tabella dando $\mathrm{ind}_{11,g}(x)$, per ogni $x \in \mathbf{Z}_{11}^*$.

33.6-2 Si dia un algoritmo di elevamento a potenza modulare che esamini i bit di b da destra a sinistra invece che da sinistra a destra.

33.6-3 Si spieghi come calcolare $a^{-1} \bmod n$ per qualche $a \in \mathbf{Z}_n^*$ usando la procedura MODULAR-EXPONENTIATION, nell'ipotesi di conoscere $\phi(n)$.

33.7 Crittografia a chiave pubblica RSA

Un sistema di crittografia a chiave pubblica può essere usato per scrivere messaggi in codice da spedire tra due parti comunicanti in modo che un ficcanaso che spia i messaggi in codice non sia in grado di decifrarli. Un sistema di crittografia a chiave pubblica abilita le due parti ad aggiungere una "firma numerica" non falsificabile alla fine del messaggio elettronico. Tale firma è la versione elettronica di una firma scritta su un documento di carta. Può essere facilmente controllata da chiunque, contraffatta da nessuno, perde la sua validità non appena qualche bit del messaggio è alterato. Quindi fornisce sia l'autentica dell'identità del mittente che del testo del messaggio firmato. È lo strumento perfetto per contratti d'affari firmati elettronicamente, controlli elettronici, ordini d'acquisto elettronici e altre comunicazioni elettroniche che devono essere autenticate.

Il sistema di crittografia a chiave pubblica RSA si basa sull'enorme differenza tra la facilità di trovare numeri primi grandi e la difficoltà di scomporre in fattori il prodotto di due numeri primi grandi. Il paragrafo 33.8 descrive una procedura efficiente per trovare numeri primi grandi e il paragrafo 33.9 discute il problema della scomposizione in fattori interi grandi.

Sistema di crittografia a chiave pubblica

In un sistema di crittografia a chiave pubblica, ogni partecipante ha una ***chiave pubblica*** e una ***chiave segreta***. Ogni chiave è un "frammento" d'informazione. Per esempio, nel sistema di crittografia RSA, ogni chiave consiste di una coppia di interi. I partecipanti "Alice" e "Bob" sono usati tradizionalmente negli esempi di crittografia; si denotano le loro chiavi pubbliche e segrete con P_A e S_A per Alice e P_B e S_B per Bob.

Ogni partecipante crea la propria chiave pubblica e quella segreta. Ognuno mantiene segreta la chiave segreta, ma può rivelare la sua chiave pubblica a chiunque e anche renderla pubblica. In realtà, è spesso conveniente assumere che la chiave pubblica di ognuno sia disponibile in una cartella pubblica, così che qualunque partecipante possa conoscere la chiave pubblica di qualunque altro partecipante.

Le chiavi pubblica e segreta specificano le funzioni che possono essere applicate a qualunque messaggio. Si denoti con D l'insieme di messaggi permessi. Per esempio, D potrebbe essere l'insieme di tutte le sequenze di bit di lunghezza finita. Si richiede che le chiavi pubblica e segreta specifichino funzioni biunivoche da D a se stesso. La funzione corrispondente alla chiave pubblica di Alice P_A è denotata con $P_A()$ e la funzione corrispondente alla sua chiave segreta S_A è denotata con $S_A()$. Le funzioni P_A e S_A sono quindi permutazioni di D. Si assume che le funzioni P_A e S_A siano calcolabili in modo efficiente data la chiave P_A o S_A corrispondente.

Le chiavi pubblica e segreta per qualunque partecipante sono una "coppia corrispondente" nel senso che specificano funzioni che sono una l'inversa dell'altra. Cioè,

$$M = S_A(P_A(M)) , \tag{33.37}$$

$$M = P_A(S_A(M)) , \tag{33.38}$$

per qualunque messaggio $M \in$ D. Trasformando M con le due chiavi P_A e S_A successivamente, in entrambi i versi, si ottiene lo stesso messaggio.

In un sistema di crittografia a chiave pubblica è essenziale che nessuno tranne Alice sia capace di calcolare la funzione $S_A()$ in una quantità di tempo ragionevole. L'inviolabilità della posta che viene scritta in codice e mandata ad Alice e l'autenticità della firma numerica di Alice si basano sull'ipotesi che Alice sia l'unica capace di calcolare $S_A()$. Questa richiesta è il motivo per cui Alice deve mantenere segreta S_A; se non lo facesse, perderebbe la sua unicità e il sistema di crittografia non potrebbe fornirle la sicurezza necessaria. L'ipotesi che solo Alice possa calcolare $S_A()$ deve valere anche se ognuno conosce P_A e può calcolare $P_A()$, la funzione inversa di S_A, in modo efficiente. La difficoltà maggiore nella progettazione di un sistema di crittografia a chiave pubblica realizzabile è capire come creare un sistema in cui si possa rivelare una trasformazione $P_A()$ senza per questo rivelare come calcolare la corrispondente trasformazione inversa $S_A()$.

In un sistema di crittografia a chiave pubblica la trasformazione in codice funziona nel modo seguente. Si supponga che Bob desideri mandare ad Alice un messaggio M in codice

Figura 33.5 *La trasformazione in codice in un sistema di crittografia a chiave pubblica. Bob trasforma il messaggio M in codice usando la chiave pubblica $P_A()$ di Alice e trasmette il risultante testo cifrato $C = P_A(M)$ ad Alice. Un ficcanaso che catturi il testo cifrato trasmesso non ottiene alcuna informazione su M. Alice riceve C e lo interpreta usando la sua chiave segreta per ottenere il messaggio originale $M = S_A(C)$.*

Figura 33.6 *Firme numeriche in un sistema di crittografia a chiave pubblica. Alice firma il messaggio M' aggiungendo in fondo la sua firma numerica $\sigma = S_A(M')$. Alice trasmette la coppia messaggio/firma (M', σ) a Bob, che la verifica utilizzando l'equazione $M' = P_A(\sigma)$. Se l'equazione è verificata, egli accetta (M', σ) come messaggio firmato da Alice.*

così che ad un ficcanaso appaia un gergo incomprensibile. Per spedire un messaggio si opera come segue.

- Bob ottiene la chiave pubblica di Alice P_A (da una cartella pubblica o direttamente da Alice).
- Bob calcola il ***testo cifrato*** $C = P_A(M)$ corrispondente al messaggio M e spedisce C ad Alice.
- Quando Alice riceve il testo cifrato C applica la sua chiave segreta S_A per recuperare il messaggio originale $M = S_A(C)$.

La figura 33.5 illustra questo processo. Poiché P_A e S_A sono funzioni inverse, Alice può calcolare M da C. Poiché solo Alice è capace di calcolare S_A, solo Alice può calcolare M da C. La trasformazione in codice di M attraverso $P_A()$ ha protetto il messaggio M dall'essere scoperto da qualcuno che non sia Alice.

Le firme numeriche sono facilmente realizzate in modo analogo in un sistema di crittografia a chiave pubblica. Si supponga che Alice desideri mandare a Bob una risposta M' firmata numericamente. Per usare la firma numerica si opera come segue.

- Alice calcola la sua ***firma numerica*** σ per il messaggio M' usando la sua chiave segreta S_A e l'equazione $\sigma = S_A(M')$.
- Alice spedisce la coppia (M', σ) messaggio/firma a Bob.
- Quando Bob riceve (M',σ), può verificare che questo proviene da Alice usando la chiave pubblica di Alice per verificare l'equazione $M' = P_A(\sigma)$. (Presumibilmente M' contiene il nome di Alice, così che Bob sappia quale chiave pubblica usare.) Se l'equazione è verificata, allora Bob conclude che il messaggio era effettivamente firmato da Alice. Se l'equazione non è verificata, Bob conclude che o il messaggio M' o la firma numerica σ si sono alterati per errore di trasmissione o che la coppia (M',σ) è un tentativo di contraffazione.

La figura 33.6 mostra questo processo. Poiché una firma numerica fornisce sia l'autentica dell'identità del firmatario che l'autentica del contenuto del messaggio firmato, essa è analoga a una firma scritta a mano alla fine di un documento di carta.

Una proprietà importante della firma numerica è che è verificabile da chiunque abbia accesso alla chiave pubblica del firmatario. Un messaggio firmato può essere verificato da un partecipante e quindi passato ad altri partecipanti che a loro volta possono verificare la firma. Per esempio il messaggio potrebbe essere un assegno elettronico da Alice a Bob. Dopo

che Bob verifica la firma di Alice sull'assegno, egli può dare l'assegno alla sua banca, la quale, a sua volta, può verificare la firma ed effettuare l'appropriato trasferimento di fondi.

Si noti che un messaggio firmato non è in codice; il messaggio è "in chiaro" e non è protetto da scoperte accidentali. Componendo i protocolli precedenti per la trasformazione in codice e per la firma, si può creare un messaggio che sia firmato e in codice. Il firmatario prima appende la sua firma numerica al messaggio e poi trasforma in codice la coppia messaggio/firma risultante tramite la chiave pubblica del destinatario desiderato. Il destinatario interpreta il messaggio ricevuto tramite la sua chiave segreta per ottenere sia il messaggio originale che la firma numerica. Può quindi verificare la firma usando la chiave pubblica del mittente firmatario. Il corrispondente processo combinato nei sistemi cartacei è di firmare il documento su carta e quindi sigillare il documento in una busta di carta che viene aperta solo dal destinatario desiderato.

Sistema di crittografia RSA

Nel ***sistema di crittografia a chiave pubblica RSA***, un partecipante crea le sue chiavi pubblica e segreta con la seguente procedura.

1. Seleziona a caso due numeri primi p e q grandi. I primi p e q potrebbero essere, per esempio, di cento cifre decimali ciascuno.
2. Calcola n dall'equazione $n = pq$.
3. Seleziona un intero piccolo dispari e tale che e e $\phi(n)$ siano primi tra loro, dove, per l'equazione (33.20), $\phi(n) = (p-1)(q-1)$.
4. Calcola d, il reciproco di e, modulo $\phi(n)$. (Il corollario 33.26 garantisce che d esiste ed è univocamente definito.)
5. Pubblica la coppia $P = (e, n)$ come sua ***chiave pubblica RSA***.
6. Tiene segreta la coppia $S = (d, n)$ come sua ***chiave segreta RSA***.

Per questo schema, il dominio D è l'insieme $\mathbf{Z}_n$. La trasformazione di un messaggio M associato ad una chiave pubblica $P = (e, n)$ è

$$P(M) = M^e \pmod n . \tag{33.39}$$

La trasformazione di un testo cifrato C associato alla chiave segreta $S = (d, n)$ è

$$S(C) = C^d \pmod n . \tag{33.40}$$

Queste equazioni si applicano alla trasformazione in codice e alle firme. Per creare una firma, il firmatario applica la sua chiave segreta al messaggio da firmare piuttosto che a un testo cifrato. Per verificare una firma, la chiave pubblica del firmatario è applicata a questo piuttosto che a un messaggio da trasformare in codice.

Le operazioni con chiave pubblica e chiave segreta possono essere realizzate usando la procedura MODULAR-EXPONENTIATION descritta al paragrafo 33.6. Per analizzare il tempo di esecuzione di queste operazioni, si assume che la chiave pubblica (e, n) e la chiave segreta (d, n) soddisfino $\lg e = O(1)$, $\lg d = \lg n = \beta$. Allora, l'applicazione di una chiave richiede $O(1)$ moltiplicazioni modulari e usa $O(\beta^2)$ operazioni sui bit. L'applicazione di una chiave segreta richiede $O(\beta)$ moltiplicazioni modulari, usando $O(\beta^3)$ operazioni sui bit.

Teorema 33.36 (Correttezza di RSA)

Le equazioni RSA (33.39) e (33.40) definiscono le trasformazioni inverse di $\mathbf{Z}_n$ che soddisfano le equazioni (33.37) e (33.38).

Dimostrazione. Dalle equazioni (33.39) e (33.40), si ha che per qualsiasi $M \in \mathbf{Z}_n$,

$$P(S(M)) = S(P(M)) = M^{ed} \pmod n ,$$

Poiché e e d sono reciproci modulo $\phi(n) = (p-1)(q-1)$,

$$ed = 1 + k(p-1)(q-1)$$

per qualche intero k. Ma allora, se $M \not\equiv 0 \pmod p$, si ha (usando il Teorema 33.31)

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} \pmod p \\ &\equiv M(1)^{k(q-1)} \pmod p \\ &\equiv M \pmod p . \end{aligned}$$

Inoltre, $M^{ed} \equiv M \pmod p$ se $M \equiv 0 \pmod p$. Pertanto

$$M^{ed} \equiv M \pmod p$$

per ogni M. Analogamente,

$$M^{ed} \equiv M \pmod q$$

per ogni M. Pertanto, per il corollario 33.29 del teorema cinese del resto,

$$M^{ed} \equiv M \pmod n$$

per ogni M. ■

La sicurezza del sistema di crittografia RSA si basa in gran parte sulla difficoltà della scomposizione in fattori di interi grandi. Se un malintenzionato può scomporre il modulo n di una chiave pubblica, allora può derivare la chiave segreta dalla chiave pubblica usando la conoscenza dei fattori p e q nello stesso modo in cui li usa il creatore della chiave pubblica. Così, se la scomposizione in fattori di interi grandi è facile, allora anche violare il sistema di crittografia RSA è facile. L'asserzione opposta, cioè che se la scomposizione in fattori di interi grandi è difficile allora anche violare il sistema RSA è difficile non è stata provata. Dopo un decennio di ricerche, però, non è stato trovato alcun metodo più facile per violare il sistema di crittografia a chiave pubblica RSA che quello di scomporre in fattori il modulo n. Come si vedrà al paragrafo 33.9, la scomposizione in fattori di numeri primi grandi è sorprendentemente difficile. Selezionando in modo casuale due primi di 100 cifre e moltiplicandoli, si può creare una chiave pubblica che, con le attuali tecnologie, non può essere "violata" in un tempo ragionevole. In assenza di nuovi e decisivi risultati nell'ambito del progetto di algoritmi per la fattorizzazione di interi, il sistema di crittografia RSA è in grado di fornire un alto grado di sicurezza alle applicazioni.

Per raggiungere la sicurezza nel sistema di crittografia RSA, però, è necessario lavorare con interi che siano lunghi 100-200 cifre, poiché la scomposizione di interi più piccoli risulta essere "ragionevolmente" computabile. In particolare, si deve essere capaci di trovare primi grandi in modo efficiente, per creare chiavi della lunghezza necessaria. Questo problema è affrontato al paragrafo 33.8.

Per motivi di efficienza, RSA è spesso usato in modo "ibrido" o "a gestione di chiavi" con veloci sistemi di crittografia a chiave non pubblica. Con un tale sistema, le chiavi per trasformare in codice e per interpretare il codice sono identiche. Se Alice desidera mandare un messaggio lungo a Bob privatamente, seleziona una chiave casuale K per il sistema di crittografia a chiave non pubblica e trasforma in codice M usando K, ottenendo il testo cifrato C. In questo caso C è lungo come M ma K è abbastanza piccola. Quindi, trasforma in codice K usando la chiave pubblica RSA di Bob. Poiché K è piccolo, calcolare $P_B(K)$ è veloce (molto più veloce che calcolare $P_B(M)$). Quindi, Alice trasmette $(C, P_B(K))$ a Bob che interpreta $P_B(K)$ per ottenere K e quindi usa K per interpretare C, ottenendo M.

Un simile approccio ibrido è spesso usato per ottenere firme numeriche in modo efficiente. In questo approccio, RSA è combinato con una ***funzione hash one-way*** h pubblica – una funzione che è facile da calcolare ma per cui è computazionalmente impossibile trovare due messaggi M e M' tali che $h(M) = h(M')$. Il valore $h(M)$ è un numero breve (per esempio 128 bit) e può essere considerato "impronta digitale" del messaggio M. Se Alice desidera firmare un messaggio M, prima applica h ad M per ottenere l'impronta digitale $h(M)$ che poi firma con la sua chiave segreta. Manda $(M, S_A(h(M)))$ a Bob come sua versione firmata del messaggio M. Bob può verificare la firma calcolando $h(M)$ e verificando che P_A applicato al messaggio $S_A(h(M))$ ricevuto è uguale a $h(M)$. Poiché nessuno può creare due messaggi con la stessa impronta digitale, è impossibile alterare un messaggio firmato e mantenere la validità della firma.

Infine si segnala l'uso di ***certificati*** che rende molto facile la distribuzione delle chiavi pubbliche. Per esempio, si assuma che vi sia una "autorità di fiducia" T la cui chiave pubblica sia conosciuta da tutti. Alice può ottenere da T un messaggio firmato (certificato di lei) che stabilisce che "la chiave pubblica di Alice è P_A". Questo certificato si autentica da sé poiché tutti conoscono P_T. Alice può includere il suo certificato ai messaggi da lei firmati, così che il destinatario abbia la chiave pubblica di Alice disponibile immediatamente in modo da poter verificare la sua firma. Poiché la sua chiave era firmata da T, il destinatario sa che la chiave di Alice è proprio la chiave di Alice.

Esercizi

33.7-1 Si consideri un insieme di chiavi RSA con $p = 11$, $n = 319$ e $e = 3$. Quale valore di d dovrebbe essere usato nella chiave segreta? Qual è il codice del messaggio $M = 100$.

33.7-2 Si dimostri che se l'esponente pubblico e di Alice è 3 e un malintenzionato ottiene l'esponente segreto d di Alice, allora il malintenzionato potrà scomporre in fattori il modulo n di Alice in tempo polinomiale rispetto al numero di bit di n. (Sebbene non sia richiesto di dimostrarlo, al lettore potrà interessare il fatto che questo risultato rimane vero anche se la condizione $e = 3$ è rimossa. Si veda Miller [147].)

* *33.7-3* Si dimostri che RSA è moltiplicativo nel senso che

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n} .$$

Si usi questo fatto per dimostrare che se un malintenzionato avesse una procedura per interpretare in modo efficiente l'1% dei messaggi casualmente scelti fra $\mathbb{Z}_n$ e trasformarli in codice con P_A, allora potrebbe impiegare un algoritmo probabilistico per interpretare con buone probabilità ogni messaggio codificato con P_A.

* 33.8 Verifica di primalità

In questo paragrafo, si considera il problema di trovare primi grandi. Si comincia con una discussione sulla densità dei primi, procedendo ad esaminare un plausibile approccio (anche se incompleto) per la verifica di primalità e quindi presentando la verifica di primalità randomizzata dovuta a Miller e Rabin.

Densità dei numeri primi

Per molte applicazioni (come la crittografia), vi è la necessità di trovare primi grandi "casuali". Fortunatamente, i primi grandi non sono troppo rari, così che non è richiesto troppo tempo per trovare un primo verificando via via interi casuali della dimensione appropriata. La ***funzione di distribuzione dei primi*** $\pi(n)$ specifica il numero di primi minori o uguali a n. Per esempio $\pi(10) = 4$, poiché vi sono 4 numeri primi minori o uguali a 10, cioè 2, 3, 5 e 7. Il teorema dei numeri primi fornisce un'utile approssimazione di $\pi(n)$.

Teorema 33.37 (Teorema dei numeri primi)

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1 . \qquad \blacksquare$$

L'approssimazione $n/\ln n$ fornisce una stima ragionevolmente accurata di $\pi(n)$ anche per piccoli n. Per esempio, dà un errore per meno del 6% per $n = 10^9$, dove $\pi(n) = 50847534$ e $n/\ln n = 48254942$. (Per un teorico dei numeri, 10^9 è un numero piccolo.)

Utilizzando il Teorema dei numeri primi, si ottiene che la probabilità che un numero n casualmente scelto sia primo è pari a $1/\ln n$. Pertanto, si dovrebbero esaminare approssimativamente $\ln n$ interi scelti casualmente vicino ad n per trovare un primo che sia della stessa lunghezza di n. Per esempio, trovare un primo di 100 cifre potrebbe richiedere la verifica di primalità di, approssimativamente, $\ln 10^{100} = 230$ numeri di 100 cifre scelti casualmente. (Questo numero può essere diviso a metà scegliendo solo interi dispari.)

Nel resto del paragrafo, si considera il problema di determinare se un intero n dispari grande è primo. Per comodità di notazione, si assuma che n abbia la seguente scomposizione in fattori:

$$n = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r} , \tag{33.41}$$

dove $r \geq 1$ e $p_1, p_2, \ldots, p_r$ sono i fattori primi di n. Naturalmente, n è primo se e solo se $r = 1$ e $e_1 = 1$.

Un semplice approccio al problema di verificare la primalità è la ***prova di divisione***. Si cerca di dividere n per ogni intero $2, 3, \ldots, \lfloor \sqrt{n} \rfloor$. (Come prima, interi pari più grandi di 2 possono essere saltati.) È facile vedere che n è primo se e solo se nessuno dei divisori provati divide n. Assumendo che ogni prova di divisione richieda tempo costante, il tempo di esecuzione è $\Theta(\sqrt{n})$ nel caso peggiore che è esponenziale rispetto alla lunghezza di n. (Si ricorda che se n è rappresentato in binario usando β bit, allora $\beta = \lceil \lg(n+1) \rceil$ e così $\sqrt{n} = \Theta(2^{\beta/2})$.) Pertanto, la prova di divisione funziona bene solo se n è molto piccolo oppure se ha un fattore primo piccolo. Quando funziona, la prova di divisione offre il vantaggio non solo di determinare se n è primo o composto ma in più, se n è composto, di determinare di fatto uno dei fattori primi di n.

In questo paragrafo, si è solamente interessati a sapere se un dato numero n è primo; se n è composto, non interessa scomporlo in fattori primi. Come si vedrà al paragrafo 33.9, il calcolo della scomposizione di un numero in fattori primi, da un punto di vista computazionale, è costoso. Può sorprendere forse il fatto che è molto più facile dire se un dato numero è primo o no che determinare la sua scomposizione in fattori primi, nel caso sia un numero composto.

Verifica di pseudoprimalità

Si consideri ora un metodo per il controllo della primalità che è "quasi corretto" e di fatto sufficientemente buono in molte applicazioni pratiche. Una raffinamento di questo metodo, che ne rimuove i piccoli difetti, sarà presentato in seguito. Si denotino con $\mathbf{Z}_n^+$ gli elementi non nulli di $\mathbf{Z}_n$:

$$\mathbf{Z}_n^+ = \{1, 2, \ldots, n-1\} .$$

Se n è primo, allora $\mathbf{Z}_n^+ = \mathbf{Z}_n^*$.

Si dice che n è uno ***pseudoprimo di base a*** se n è composto e

$$a^{n-1} \equiv 1 \pmod{n} . \tag{33.42}$$

Il Teorema di Fermat (Teorema 33.31) implica che se n è primo, allora n soddisfa l'equazione (33.42) per ogni a in $\mathbf{Z}_n^+$. Pertanto, se si può trovare qualche $a \in \mathbf{Z}_n^+$ tale che n *non* soddisfi l'equazione (33.42), allora n è certamente composto. Sorprendentemente, il viceversa è *quasi* valido, così che questo criterio definisce una verifica di primalità quasi perfetta. Si verifica se n soddisfa l'equazione (33.42) per $a = 2$. Se è falsa, si dichiara che n è composto. Altrimenti, si spera che n sia primo (anche se, in effetti, si sa che n è o primo o uno pseudoprimo di base 2).

La seguente procedura simula in questo modo il controllo della primalità di n. Essa usa la procedura MODULAR-EXPONENTIATION del paragrafo 33.6. Si assume che l'input n sia un intero maggiore di 2.

```
PSEUDOPRIME(n)
if MODULAR-EXPONENTIATION(2, n - 1, n) ≢ 1 (mod n)
  then return COMPOSTO      ▷ Di sicuro.
  else return PRIMO         ▷ Si spera!
```

Questa procedura può produrre degli errori, ma solo di un tipo. Cioè, se dice che n è composto, allora è sempre corretta. Se dice che n è primo, però, fa un errore solo se n è uno pseudoprimo di base 2.

Quanto spesso questa procedura dà risultati sbagliati? Raramente. Vi sono solo 22 valori di n minori di 10000 per cui sbaglia; i primi quattro di tali valori sono 341, 561, 645 e 1105. Si può mostrare che la probabilità che questo programma faccia un errore su un numero scelto a caso di β bit, tende a zero per $\beta \to \infty$. Usando una stima più precisa dovuta a Pomerance [157] del numero di pseudoprimi di base 2 di una data dimensione, si può stimare che un numero di 50 cifre scelto a caso dichiarato primo dalla procedura ha meno di una possibilità su un milione di essere uno pseudoprimo di base 2, e un numero di 100 cifre scelto a caso dichiarato primo ha meno di una possibilità su 10^{13} di essere uno pseudoprimo di base 2.

Sfortunatamente, non si possono eliminare tutti gli errori controllando semplicemente l'equazione (33.42) per un secondo numero base, come $a = 3$, perché vi sono interi composti che soddisfano l'equazione (33.42) per *tutte le* $a \in \mathbf{Z}_n^*$. Questi interi sono conosciuti come ***numeri di Carmichael***. I primi tre numeri di Carmichael sono 561, 1105 e 1729. I numeri di Carmichael sono estremamente rari; per esempio, quelli più piccoli di 100000000 sono solo 255. L'Esercizio 33.8-2 aiuta a spiegare perché sono così rari.

Si mostrerà in seguito come migliorare la verifica di primalità proposta in modo che essa non fallisca per i numeri di Carmichael.

Verifica di primalità randomizzata di Miller-Rabin

La verifica di primalità di Miller-Rabin supera il problema della semplice verifica PSEUDOPRIME con due modifiche:

- Prova alcuni valori a di base scelti in modo casuale invece di un solo valore di base.
- Mentre calcola ogni elevamento a potenza modulare, rileva se ha scoperto una radice quadrata non banale di 1 modulo n. Se è così, si ferma e restituisce COMPOSTO. Il corollario 33.35 giustifica questa ricerca dei composti.

Lo pseudocodice per il controllo di primalità di Miller-Rabin viene dato di seguito. L'input $n > 2$ è il numero dispari su cui fare la verifica di primalità e s è il numero di valori di base da provare, scelti casualmente da $\mathbf{Z}_n^+$. Il codice usa il generatore di numeri casuali RANDOM del paragrafo 8.3: RANDOM$(1, n-1)$ restituisce un intero a scelto casualmente che soddisfa $1 \le a \le n-1$. Il codice usa una procedura ausiliaria WITNESS tale che WITNESS(a, n) è TRUE se e solo se a è un "testimone" del fatto che n è composto – cioè è possibile usare a per dimostrare (come si vedrà) che n è composto. Il controllo WITNESS(a, n) è simile, ma più efficace, del controllo

$$a^{n-1} \not\equiv 1 \pmod{n}$$

che è stato usato come base per PSEUDOPRIME (con $a = 2$). Si presenta e si giustifica dapprima la costruzione di WITNESS, quindi si mostra come la procedura viene usata nella verifica di primalità di Miller-Rabin.

```
WITNESS(a,n)
Sia ⟨b_k, b_{k-1}, ..., b_0⟩ la rappresentazione binaria di n - 1
d ← 1
for i ← k downto 0
    do x ← d
       d ← (d · d) mod n
       if d = 1 e x ≠ 1 e x ≠ n - 1
          then return TRUE
       if b_i = 1
          then d ← (d · a) mod n
if d ≠ 1
  then return TRUE
return FALSE
```

Questo pseudocodice per WITNESS è basato sullo pseudocodice della procedura MODULAR-EXPONENTIATION. La linea 1 determina la rappresentazione binaria di $n-1$ che sarà usata per elevare a alla $(n-1)$-esima potenza. Le linee 3-9 calcolano d come $a^{n-1} \pmod n$. Il metodo usato è identico a quello impiegato da MODULAR-EXPONENTIATION. Ogni volta che alla linea 5 è eseguito l'elevamento al quadrato, si controlla alle linee 6-7 se è stata appena trovata una radice quadrata non banale di 1. In tal caso, l'algoritmo si ferma e restituisce TRUE. Le linee 10-11 restituiscono TRUE se il valore calcolato per $a^{n-1} \pmod n$ non è uguale a 1, proprio come la procedura PSEUDOPRIME in questo caso restituisce COMPOSTO.

Si mostra ora che se WITNESS(a,n) restituisce TRUE, allora, usando a, è possibile provare che n è composto.

Se WITNESS restituisce TRUE alla linea 11, allora la procedura ha scoperto che $d = a^{n-1} \pmod n \neq 1$. Se n è primo, però, si ha dal Teorema di Fermat (Teorema 33.31) che $a^{n-1} \equiv 1 \pmod n$ per tutte le $a \in \mathbf{Z}_n^*$. Quindi n non può essere primo, e l'equazione $a^{n-1} \bmod n \neq 1$ è una prova di questo fatto.

Se WITNESS restituisce TRUE alla linea 7, allora la procedura ha scoperto che x è una radice quadrata non banale di 1 modulo n, poiché si ha che $x \not\equiv \pm 1 \pmod n$ e anche $x^2 \equiv 1 \pmod n$. Il Corollario 33.35 stabilisce che solo se n è composto ci può essere una radice quadrata non banale di 1 modulo n, così che la dimostrazione che x è una radice quadrata non banale di 1 modulo n è una prova che n è composto.

Con ciò la dimostrazione della correttezza di WITNESS è completa. Se la chiamata di WITNESS(a, b) dà come risultato TRUE, allora n è sicuramente composto e una dimostrazione che n è composto può essere facilmente costruita usando a e n. Si esamina ora la verifica di primalità di Miller-Rabin basata sull'uso di WITNESS

```
MILLER-RABIN(n, s)
1  for j ← 1 to s
2      do a ← RANDOM(1, n − 1)
3         if WITNESS(a, n)
4            then return COMPOSTO        ▷ Di sicuro.
5  return PRIMO                          ▷ Quasi sicuramente.
```

La procedura MILLER-RABIN è una ricerca probabilistica di una dimostrazione che n sia composto. Il ciclo principale (a cominciare dalla linea 1) prende s valori casuali di a da $\mathbf{Z}_n^*$ (linea 2). Se una delle a scelte porta a dire che n è composto, allora MILLER-RABIN restituisce COMPOSTO alla linea 4. Questo risultato è sempre corretto, data la correttezza di WITNESS. Se nessun testimone del fatto che n è composto può essere trovato in s prove, MILLER-RABIN assume che ciò accade perché non vi sono testimoni da trovare e quindi n è primo. Si vedrà che questo risultato è probabile che sia corretto se s è sufficientemente grande, ma vi è una piccola possibilità che la procedura possa essere sfortunata nella scelta delle a e che qualche testimone esista anche se non ne è stato trovato alcuno.

Per mostrare il funzionamento di MILLER-RABIN, sia n il numero di Carmichael 561. Supponendo che $a = 7$ sia scelto come base, la figura 33.4 mostra che WITNESS scopre una radice quadrata non banale di 1 nell'ultimo passo di elevamento al quadrato, poiché

$a^{280} \equiv 67 \pmod n$ e $a^{560} \equiv 1 \pmod n$. Allora, $a = 7$ è un testimone del fatto che n è composto. WITNESS$(7, n)$ restituisce TRUE e MILLER-RABIN restituisce COMPOSTO.

Se n è un numero di β bit, MILLER-RABIN richiede $O(s\beta)$ operazioni aritmetiche e $O(s\beta^3)$ operazioni sui bit, poiché asintoticamente l'operazione più onerosa risulta essere l'elevamento a potenza modulare.

Tasso di errore del test di primalità di Miller-Rabin

Se MILLER-RABIN restituisce PRIMO, allora vi è una piccola possibilità che abbia commesso un errore. Diversamente da PSEUDOPRIME, però, la possibilità di errore non dipende da n: non vi sono input "cattivi" per questo algoritmo. Piuttosto, dipende da quanto è grande s e dalla fortuna nella scelta dei valori di base a. Inoltre, poiché ogni controllo è più stringente di un semplice controllo dell'equazione (33.42), ci si può aspettare in generale che il tasso di errore debba essere piccolo per interi n scelti a caso. Il seguente teorema presenta un'argomentazione più precisa.

Teorema 33.38

Se n è un numero dispari composto, allora il numero di testimoni del fatto che n è composto è almeno $(n-1)/2$.

Dimostrazione. Si mostra che il numero di non testimoni non è più di $(n-1)/2$, da cui segue il teorema.

Prima si osserva che qualunque non testimone deve essere un elemento di $\mathbf{Z}_n^*$ poiché ogni non testimone a soddisfa $a^{n-1} \equiv 1 \pmod n$ e se $\mathrm{MCD}(a,n) = d > 1$, allora non vi sono soluzioni x all'equazione $ax \equiv 1 \pmod n$, dal corollario 33.21. (In particolare, $x = a^{n-2}$ non è una soluzione.) Pertanto ogni elemento di $\mathbf{Z}_n - \mathbf{Z}_n^*$ è un testimone del fatto che n è composto.

Per completare la dimostrazione, si mostra che i non testimoni sono tutti contenuti in un sottogruppo proprio B di $\mathbf{Z}_n^*$. Dal corollario 33.16 allora si ha $|B| \le |\mathbf{Z}_n^*|/2$. Poiché $|\mathbf{Z}_n^*| \le n-1$, si ottiene $|B| \le (n-1)/2$. Pertanto, il numero di non testimoni deve essere almeno $(n-1)/2$.

Si mostra ora come trovare un sottogruppo proprio di $\mathbf{Z}_n^*$ contenente tutti i non testimoni. Si spezza la dimostrazione in due casi.

Caso 1: Esiste un $x \in \mathbf{Z}_n^*$ tale che

$$x^{n-1} \not\equiv 1 \pmod n . \tag{33.43}$$

Sia $B = \{b \in \mathbf{Z}_n^* : b^{n-1} \equiv 1 \pmod n\}$. Poiché B è chiuso rispetto alla moltiplicazione modulo n, si ha che B è un sottogruppo di $\mathbf{Z}_n^*$ per il Teorema 33.14. Si noti che ogni non testimone appartiene a B, poiché un non testimone a soddisfa $a^{n-1} \equiv 1 \pmod n$. Poiché $x \in \mathbf{Z}_n^* - B$, si ha che B è un sottogruppo proprio di $\mathbf{Z}_n^*$.

Caso 2: Per tutte le $x \in \mathbf{Z}_n^*$,

$$x^{n-1} \equiv 1 \pmod n . \tag{33.44}$$

In questo caso, n non può essere potenza di un primo. Per vederne il motivo, sia $n = p^e$, dove p è un primo dispari ed $e > 1$. Il Teorema 33.32 implica che $\mathbf{Z}_n^*$ contenga un elemento g tale che $\mathrm{ord}_n(g) = |\mathbf{Z}_n^*| = \phi(n) = (p-1)p^{e-1}$. Ma allora l'equazione (33.44) e il teorema del logaritmo discreto (Teorema 33.33, con $y = 0$) implicano che $n - 1 \equiv 0 \pmod{\phi(n)}$.

oppure

$$(p-1)p^{e-1} \mid p^e - 1 .$$

Questa condizione fallisce per $e > 1$, poiché in tal caso il lato sinistro è divisibile per p ma il lato destro non lo è. Pertanto, n non è potenza di un primo.

Poiché n non è potenza di un primo, esso si scompone in un prodotto $n_1 n_2$, dove n_1 e n_2 sono più grandi di 1 e primi tra loro. (Vi possono essere diversi modi per fare ciò e non importa quale si sceglie. Per esempio, se $n = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$, allora si può scegliere $n_1 = p_1^{e_1}$ e $n_2 = p_2^{e_2} p_3^{e_3} \ldots p_r^{e_r}$.)

Si definiscono t ed u tali che $n - 1 = 2^t u$, dove $t \geq 1$ e u è dispari. Per qualsiasi $a \in \mathbf{Z}_n^*$, si consideri la sequenza

$$\hat{a} = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle , \tag{33.45}$$

dove tutti gli elementi sono calcolati modulo n. Poiché $2^t \mid n - 1$, la rappresentazione binaria di $n - 1$ termina con t zeri e gli elementi di $\hat{a}$ sono gli ultimi $t + 1$ valori di d calcolati da WITNESS durante un calcolo di $a^{n-1} \bmod n$; le ultime t operazioni sono di elevamento al quadrato.

Ora si trovi $j \in \{0, 1, \ldots, t\}$ per cui esista una $v \in \mathbf{Z}_n^*$ tale che $v^{2^j u} \equiv -1 \pmod{n}$; j dovrebbe essere il più grande possibile. Un tale j certamente esiste poiché u è dispari: si può scegliere $v = -1$ e $j = 0$. Si fissi v per soddisfare la condizione data. Sia

$$B = \{x \in \mathbf{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\} .$$

Essendo chiuso rispetto alla moltiplicazione modulo n, B è un sottogruppo di $\mathbf{Z}_n^*$. Di conseguenza, $|B|$ divide $|\mathbf{Z}_n^*|$. Ogni non testimone deve essere un elemento di B, dato che la sequenza (33.45) prodotta da un non testimone deve essere o di tutti 1 o altrimenti contenere un -1 non oltre la j-esima posizione, per la massimalità di j.

Si usa ora l'esistenza di v per dimostrare che esiste un $w \in \mathbf{Z}_n^* - B$. Poiché $v^{2^j u} \equiv -1 \pmod{n}$, si ha $v^{2^j u} \equiv -1 \pmod{n_1}$ per il corollario 33.29. Dal corollario 33.28, vi è un w che soddisfa contemporaneamente le equazioni

$$w \equiv v \pmod{n_1} ,$$

$$w \equiv 1 \pmod{n_2} .$$

Allora,

$$w^{2^j u} \equiv -1 \pmod{n_1} ,$$

$$w^{2^j u} \equiv 1 \pmod{n_2} .$$

Insieme con il corollario 33.29, queste equazioni implicano che

$$w^{2^j u} \not\equiv \pm 1 \pmod{n} , \tag{33.46}$$

e così $w \notin B$. Poiché $v \in \mathbf{Z}_n^*$, si ha che $v \in \mathbf{Z}_{n_1}^*$. Pertanto $w \in \mathbf{Z}_n^*$ e così $w \in \mathbf{Z}_n^* - B$. Si conclude che B è un sottogruppo proprio di $\mathbf{Z}_n^*$.

In ogni caso, si vede che il numero di testimoni del fatto che n è composto è almeno $(n-1)/2$. ■

Teorema 33.39

Per un qualunque intero dispari $n > 2$ e un intero s positivo, la probabilità che MILLER-RABIN(n, s) dia un errore è al più 2^{-s}.

Dimostrazione. Usando il Teorema 33.38, si vede che se n è composto, allora ogni esecuzione del ciclo alle linee 1-4 ha probabilità almeno 1/2 di scoprire un testimone x del fatto che n è composto. La procedura MILLER-RABIN fa un errore solo se è così sfortunata da non scoprire un testimone in ognuna delle s iterazioni del ciclo principale. La probabilità di una tale sequenza di insuccessi è al più 2^{-s}. ■

Pertanto, la scelta di $s = 50$ dovrebbe essere sufficiente quasi per qualsiasi applicazione immaginabile. Se si sta cercando di trovare numeri primi grandi applicando la MILLER-RABIN a interi grandi *scelti in modo casuale*, allora si può dimostrare (sebbene non si voglia farlo qui) che scegliendo un valore piccolo di s (come 3) è molto improbabile ottenere risposte erronee. Cioè, per un intero n dispari composto scelto casualmente, il numero medio di non testimoni del fatto che n è composto è probabile che sia molto più piccolo di $(n-1)/2$. Se l'intero n non è scelto in modo casuale però, il meglio che si possa dimostrare è che il numero di non testimoni è al più $(n-1)/4$, usando una versione migliorata del Teorema 33.39. Inoltre, esistono interi n per cui il numero di non testimoni è $(n-1)/4$.

Esercizi

33.8-1 Si dimostri che, se un intero dispari $n > 1$ non è primo né potenza di un primo, allora esiste una radice quadrata non banale di 1 modulo n.

* *33.8-2* È possibile definire il Teorema di Eulero in modo leggermente più forte nella forma $a^{\lambda(n)} \equiv 1 \pmod{n}$ per tutte le $a \in \mathbf{Z}_n^*$,

dove $\lambda(n)$ è definita come

$$\lambda(n) = \text{mcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})) . \tag{33.47}$$

Si dimostri che $\lambda(n) \mid \phi(n)$. Un numero composto n è un numero di Carmichael se $\lambda(n) \mid n - 1$. Il più piccolo numero di Carmichael è $561 = 3 \cdot 11 \cdot 17$; in questo caso, $\lambda(n) = \text{mcm}(2, 10, 16) = 80$, che divide 560. Si dimostri che i numeri di Carmichael devono essere sia "liberi dal quadrato" (non divisibili per il quadrato di qualsiasi primo) sia il prodotto di almeno tre primi. Per questa ragione, essi non sono molto comuni.

33.8-3 Si dimostri che se x è una radice quadrata non banale di 1 modulo n, allora $\text{MCD}(x - 1, n)$ e $\text{MCD}(x + 1, n)$ sono entrambi divisori non banali di n.

* 33.9 Scomposizione di interi in fattori primi

Si supponga di avere un intero n da *scomporre in fattori primi*, cioè da scomporre in un prodotto di numeri primi. La verifica di primalità del paragrafo precedente dice solo che n è composto, ma di solito non fornisce i fattori primi di n. Scomporre in fattori un intero grande sembra essere molto più difficile che determinare semplicemente se n è primo o composto. A tutt'oggi è praticamente impossibile con i supercalcolatori moderni e il miglior algoritmo scomporre in fattori primi un numero arbitrario di 200 cifre decimali.

Euristica rho di Pollard

La prova di divisione per tutti gli interi fino a B garantisce di scomporre completamente qualunque numero fino a B^2. Con la stessa quantità di lavoro la seguente procedura scomporrà qualsiasi numero fino a B^4 (a meno che non si sia sfortunati). Poiché la procedura è solo un'euristica, il suo tempo di esecuzione e il suo successo non sono garantiti; essa tuttavia è molto efficace in pratica.

```
POLLARD-RHO(n)
1  i ← 1
2  x_1 ← RANDOM(0,n − 1)
3  y ← x_1
4  k ← 2
5  while TRUE
6      do i ← i + 1
7          x_i ← (x_{i−1}^2 − 1) mod n
8          d ← MCD(y − x_i,n)
9          if d ≠ 1 e d ≠ n
10          then stampa d
11          if i = k
12          then y ← x_i
13               k ← 2k
```

La procedura funziona come segue. Le linee 1-2 inizializzano i a 1 e x_1 a un valore scelto casualmente in $\mathbf{Z}_n$. Il ciclo **while** che comincia alla linea 5 si ripete all'infinito, alla ricerca dei fattori di n. Durante ogni iterazione del ciclo **while**, la ricorrenza

$$x_i \leftarrow (x_{i-1}^2 - 1) \bmod n \tag{33.48}$$

è usata alla linea 7 per produrre il prossimo valore di x_i nella sequenza infinita

$$x_1, x_2, x_3, x_4, \ldots; \tag{33.49}$$

il valore di i è incrementato corrispondentemente alla linea 6. Il codice è scritto usando variabili x_i per chiarezza, ma il programma funzionerebbe ugualmente se tutti gli indici alle variabili fossero omessi, poiché solo il valore più recente di x_i deve essere mantenuto.

Di quando in quando, il programma salva nella variabile y il valore di x_i generato più recentemente. In particolare, i valori che vengono salvati sono quelli i cui indici sono potenze di 2:

$$x_1, x_2, x_4, x_8, x_{16}, \ldots .$$

Alla linea 3 si salva il valore x_1 e alla linea 12 si salva x_k se i è uguale k. La variabile k è inizializzata a 2 alla linea 4 e viene raddoppiata alla linea 13 se y viene aggiornata. Quindi, k segue la sequenza 1, 2, 4, 8 e dà sempre l'indice del prossimo valore x_k da salvare in y.

Alle linee 8-10 si cerca di trovare un fattore di n usando il valore salvato di y e il valore corrente di x_i. In particolare, alla linea 8 si calcola il massimo comun divisore $d = \text{MCD}(y - x_i, n)$. Se d è un divisore non banale di n (controllato alla linea 9), allora alla linea 10 si stampa d.

Figura 33.7 *Euristica rho di Pollard.* (a) *I valori prodotti dalla ricorrenza* $x_{i+1} \leftarrow (x_i^2 - 1) \bmod 1387$ *a partire da* $x_1 = 2$. *La scomposizione in fattori primi di* 1387 *è* 19·73. *Le frecce più grosse indicano i passi di iterazione che sono eseguiti prima che sia scoperto il fattore* 19. *Le frecce più sottili puntano a valori non raggiunti nella iterazione, per illustrare la forma a "rho". I valori ombreggiati sono i valori* y *memorizzati da* POLLARD-RHO. *Il fattore* 19 *è scoperto dopo che è stato raggiunto* $x_7 = 177$, *quando viene calcolato* MCD(63-177, 1387) = 19. *Il primo valore* x *che dovrebbe essere ripetuto è* 1186, *ma il fattore* 19 *è scoperto prima che questo valore venga raggiunto.* (b) *I valori prodotti dalla stessa ricorrenza modulo* 19. *Ogni valore* x_i *dato nella parte* (a) *è congruente modulo* 19 *al valore* x'_i *mostrato qui. Per esempio, sia* $x_4 = 63$ *che* $x_7 = 177$ *sono congruenti a* 6 *modulo* 19. (c) *I valori prodotti dalla stessa ricorrenza modulo* 73. *Ogni valore* x_i *dato nella parte* (a) *è congruente modulo* 73 *al valore* x''_i *mostrato qui. Dal Teorema cinese del resto, ogni nodo nella parte* (a) *corrisponde a una coppia di nodi, uno della parte* (b) *e uno della parte* (c).

Questa procedura per trovare un fattore all'inizio può sembrare in qualche modo misteriosa. Si noti, però, che POLLARD-RHO non stampa mai una risposta scorretta; qualunque numero stampi è un divisore non banale di n. Tuttavia POLLARD-RHO potrebbe non stampare assolutamente nulla; non ci sono garanzie che produrrà qualche risultato. Si vedrà però che vi è una buona ragione per aspettarsi che POLLARD-RHO stampi un fattore p di n approssimativamente dopo $\sqrt{p}$ iterazioni del ciclo **while**. Pertanto se n è composto, ci si può aspettare che questa procedura scopra abbastanza divisori da scomporre n completamente all'incirca dopo $n^{1/2}$ aggiornamenti, poiché ogni fattore primo p di n, tranne eventualmente il più grande, è minore di $\sqrt{n}$.

Si analizza il comportamento di questa procedura studiando quanto debba essere lunga una sequenza casuale modulo n perché un valore si ripeta. Poiché $\mathbf{Z}_n$ è finito e poiché ogni valore nella sequenza (33.49) dipende solo dal valore precedente, la sequenza (33.49) può ripetersi. Una volta raggiunto un x_i tale che $x_i = x_j$ per qualche $j < i$, ci si trova in un ciclo, poiché $x_{i+1} = x_{j+1}$, $x_{i+2} = x_{j+2}$ e così via. La ragione del nome "euristica rho" è che, come mostra la

figura 33.7, la sequenza $x_1, x_2, \ldots, x_{j-1}$ può essere vista come la "coda" del ρ e il ciclo $x_j, x_{j+1}, x_{j+2}, \ldots, x_i$ come il "corpo" del ρ.

Si studia ora il tempo richiesto perché la sequenza x_i si ripeta. Questo non è esattamente ciò di cui si ha bisogno, ma si vedrà in seguito come modificare l'argomentazione.

Avendo come obiettivo questa stima, si assuma che la funzione $(x^2 - 1) \bmod n$ si comporti come una funzione "casuale". Naturalmente non è realmente casuale, ma questa ipotesi porta a risultati consistenti con il comportamento osservato di POLLARD-RHO. Si può allora considerare ogni x_i come se fosse stato preso indipendentemente da $\mathbf{Z}_n$ secondo una distribuzione uniforme di $\mathbf{Z}_n$. Dall'analisi del paradosso dei compleanni del paragrafo 6.6.1, il numero medio di passi fatti prima che la sequenza vada in ciclo è $\Theta(\sqrt{n})$.

Si esaminano ora le modifiche richieste. Sia p un fattore non banale di n tale che $\mathrm{MCD}(p, n/p) = 1$. Per esempio, se n ha la scomposizione $n = p_1^{e_1} p_2^{e_2} \ldots p_r^{e_r}$, allora si può prendere $p = p_1^{e_1}$. (Se $e_1 = 1$, allora p è proprio il più piccolo fattore primo di n, un buon esempio da tenere presente). La sequenza $\langle x_i \rangle$ induce una sequenza corrispondente $\langle x_i' \rangle$ modulo p, dove

$$x_i' = x_i \bmod p$$

per tutte le i. Inoltre, segue dal Teorema cinese del resto che

$$x_{i+1}' = (x_i'^2 - 1) \bmod p \tag{33.50}$$

poiché

$$(x \bmod n) \bmod p = x \bmod p$$

per l'Esercizio 33.1-6.

Ragionando come prima, si trova che il numero medio di passi prima che la sequenza $\langle x_i' \rangle$ si ripeta è $\Theta(\sqrt{p})$. Se p è piccolo rispetto a n, la sequenza $\langle x_i' \rangle$ si può ripetere molto più rapidamente della sequenza $\langle x_i \rangle$. Infatti, la sequenza $\langle x_i' \rangle$ si ripete non appena due elementi della sequenza $\langle x_i \rangle$ siano congruenti modulo p, piuttosto che modulo n. Si veda la figura 33.7, parti (b) e (c), per una spiegazione.

Si denoti con t l'indice del primo valore ripetuto nella sequenza $\langle x_i' \rangle$ e con $u > 0$ la lunghezza del ciclo che è stato così prodotto. Cioè t e $u > 0$ sono i valori più piccoli tali che $x_{t+i}' = x_{t+u+i}'$ per ogni $i \geq 0$. Dalle argomentazioni precedenti, i valori medi di t e u sono entrambi $\Theta(\sqrt{p})$. Si noti che se $x_{t+i}' = x_{t+u+i}'$ allora $p \mid (x_{t+u+i} - x_{t+i})$. Pertanto, $\mathrm{MCD}(x_{t+u+i} - x_{t+i}, n) > 1$.

Quindi, una volta che POLLARD-RHO abbia salvato in y qualunque valore di x_k con $k \geq t$, allora $y \bmod p$ è sempre sul ciclo modulo p. (Se un nuovo valore è salvato in y, anche quel valore è sul ciclo modulo p.) Eventualmente, a k è assegnato un valore che è più grande di u e la procedura allora fa un intero giro attorno al ciclo modulo p senza cambiare il valore di y. Un fattore di n viene quindi scoperto quando x_i "incorre" nel valore y modulo p, precedentemente memorizzato, cioè quando $x_i \equiv y \pmod{p}$.

Presumibilmente, il fattore trovato è il fattore p, sebbene possa occasionalmente accadere che venga scoperto un multiplo di p. Poiché i valori attesi di t e di u sono $\Theta(\sqrt{p})$, il numero medio di passi richiesti per produrre il fattore p è $\Theta(\sqrt{p})$.

Vi sono due ragioni per cui questo algoritmo potrebbe non comportarsi come ci si aspetta. Primo, l'analisi euristica del tempo di esecuzione non è rigorosa ed è possibile che il ciclo di valori modulo p, possa essere più grande di $\sqrt{p}$. In questo caso l'algoritmo si comporta correttamente ma molto più lentamente di quanto richiesto. In pratica, questo sembra non essere un problema. In secondo luogo, i divisori di n prodotti da questo algoritmo potrebbero

essere sempre uno dei fattori banali 1 o n. Per esempio si supponga che $n = pq$, dove p e q sono primi. Può accadere che i valori di t e u per p siano identici ai valori di t e u per q e pertanto il fattore p è identificato sempre dalla stessa operazione di MCD che identifica il fattore q. Poiché entrambi i fattori vengono identificati allo stesso tempo, viene identificato il fattore banale $pq = n$, il che è inutile. Anche questo sembra non essere un problema reale in pratica. Se necessario, l'euristica può ricominciare con una ricorrenza diversa della forma $x_{i+1} \leftarrow (x_i^2 - c) \bmod n$. (I valori $c = 0$ e $c = 2$ dovrebbero essere evitati per ragioni che non approfondiremo qui, ma ci sono molti altri valori adatti.)

Naturalmente questa analisi è euristica e non rigorosa, poiché la ricorrenza non è veramente "casuale". Tuttavia, la procedura si comporta bene in pratica e sembra essere efficiente come indicato dall'analisi euristica. È il metodo di scelta per trovare fattori primi piccoli di un numero grande. Per scomporre completamente un numero composto n di β bit si ha bisogno di trovare solo tutti i fattori primi minori di $\lfloor n^{1/2} \rfloor$ e così ci si aspetta che POLLARD-RHO richieda al più $n^{1/4} = 2^{\beta/4}$ operazioni aritmetiche e al più $n^{1/4}\beta^3 = 2^{\beta/4}\beta^3$ operazioni sui bit. La capacità di POLLARD-RHO di trovare un fattore piccolo p di n con un numero medio $\Theta(\sqrt{p})$ di operazioni aritmetiche è spesso la sua caratteristica più attraente.

Esercizi

33.9-1 In riferimento al tracciato dell'esecuzione di POLLARD-RHO mostrata nella figura 33.7(a), quando è stampato il fattore 73 di 1387 dalla procedura?

33.9-2 Si supponga che siano dati una funzione $f: \mathbf{Z}_n \to \mathbf{Z}_n$ e un valore iniziale $x_0 \in \mathbf{Z}_n$. Si definisca $x_i = f(x_{i-1})$ per $i = 1, 2, \ldots$. Siano t e $u > 0$ i valori più piccoli tali che $x_{t+i} = x_{t+u+i}$ per $i = 0, 1, \ldots$. Nella terminologia dell'algoritmo rho di Pollard, t è la lunghezza della coda e u è la lunghezza del ciclo del rho. Si dia un algoritmo efficiente per determinare t e u in modo esatto e si analizzi il suo tempo d'esecuzione.

33.9-3 Quanti passi ci si aspettano perché POLLARD-RHO scopra un fattore nella forma p^e, dove p è primo ed $e > 1$?

* *33.9-4* Uno svantaggio di POLLARD-RHO così come è scritta è che richiede un calcolo del MCD per ogni passo della ricorrenza. È stato suggerito che si potrebbe concentrare il calcolo del MCD accumulando il prodotto di alcune x_i in una riga e quindi prendere il MCD di questo prodotto con la y salvata. Si descriva in modo preciso come si potrebbe realizzare quest'idea, giustificandone la correttezza. Qual è il numero più opportuno di x_i da considerare se si opera con un numero n di β bit?

Problemi

33-1 *Algoritmo del MCD binario*

Su molti calcolatori, le operazioni di sottrazione, controllo di parità (pari o dispari) di un intero binario e divisione per 2 possono essere eseguite più rapidamente del calcolo dei resti.

Questo problema studia ***l'algoritmo del MCD binario*** che evita i calcoli del resto usati nell'algoritmo di Euclide.

a. Si dimostri che se a e b sono entrambi pari, allora $\text{MCD}(a, b) = 2\text{MCD}(a/2, b/2)$.

b. Si dimostri che se a è dispari e b è pari, allora $\text{MCD}(a, b) = \text{MCD}(a, b/2)$.

c. Si dimostri che se a e b sono entrambi dispari, allora $\text{MCD}(a, b) = \text{MCD}((a - b)/2, b)$.

d. Si progetti un algoritmo efficiente per il MCD binario di due interi a e b in input, con $a \geq b$, che venga eseguito in tempo $O(\lg(\max(a, b)))$. Si assuma che ogni operazione di sottrazione, controllo di parità e divisione per 2 possa essere eseguita in un'unità di tempo.

33-2 *Analisi delle operazioni sui bit dell'algoritmo di Euclide*

a. Si mostri che il comune algoritmo "carta e penna" per divisioni lunghe – la divisione di a per b, ottenendo quoziente q e resto r – richiede $O((1 + \lg q)\lg b)$ operazioni sui bit.

b. Si definisca $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Si mostri che il numero di operazioni sui bit eseguite dall'algoritmo di Euclide nel ridurre il problema di calcolare $\text{MCD}(a, b)$ a quello di calcolare $\text{MCD}(b, a \bmod b)$ è al più $c(\mu(a, b) - \mu(b, a \bmod b))$, per qualche costante $c > 0$ sufficientemente grande.

c. Si mostri che $\text{EUCLID}(a, b)$ richiede $O(\mu(a, b))$ operazioni sui bit in generale e $O(\beta^2)$ operazioni sui bit quando è applicato a due input di β bit.

33-3 *Tre algoritmi per i numeri di Fibonacci*

Questo problema confronta l'efficienza di tre metodi per calcolare l'n-esimo numero di Fibonacci F_n, dato n. Si assuma che il costo di sommare, sottrarre, o moltiplicare due numeri sia $O(1)$, indipendentemente dalla grandezza dei numeri.

a. Si mostri che il tempo di esecuzione del metodo ricorsivo diretto per calcolare F_n basato sulla ricorrenza (2.13) è esponenziale rispetto a n.

b. Si mostri come calcolare F_n in tempo $O(n)$ usando la ricorsione con memorizzazione.

c. Si mostri come calcolare F_n in tempo $O(\lg n)$ usando solo addizione e moltiplicazione tra interi. (*Suggerimento:* si consideri la matrice

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

e le sue potenze.)

d. Si assuma ora che la somma di due numeri di β bit richieda tempo $\Theta(\beta)$ e che la moltiplicazione di due numeri di β bit richieda tempo $\Theta(\beta^2)$.

Qual è il tempo di esecuzione di questi tre metodi con questa misura, più ragionevole, dei costi delle operazioni aritmetiche elementari ?

33-4 *Residui quadratici*

Sia p un primo dispari. Un numero $a \in \mathbf{Z}_p^*$ è un ***residuo quadratico*** se l'equazione $x^2 \equiv a \pmod p$ ha una soluzione nell'incognita x.

a. Si mostri che vi sono esattamente $(p - 1)/2$ residui quadratici modulo p.

b. Se p è primo si definisce, per $a \in \mathbf{Z}_p^*$, il ***simbolo di Legendre*** $\left(\frac{a}{p}\right)$ uguale a 1 se a è un residuo quadratico modulo p e -1 altrimenti. Si provi che se $a \in \mathbf{Z}_p^*$, allora

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod p \ .$$

Si dia un algoritmo efficiente per determinare se un dato numero a è un residuo quadratico modulo p. Si analizzi l'efficienza dell'algoritmo.

c. Si dimostri che se p è un primo della forma $4k + 3$ e a è un residuo quadratico in $\mathbf{Z}_p^*$, allora $a^{k+1} \bmod p$ è una radice quadrata di a modulo p. Quanto tempo è richiesto per trovare la radice quadrata di un residuo quadratico a modulo p?

d. Si descriva un algoritmo randomizzato efficiente per trovare un residuo non quadratico modulo un arbitrario numero primo p. Quante operazioni aritmetiche richiede in media l'algoritmo trovato?

Note al capitolo

Niven e Zuckerman [151] forniscono un'ottima introduzione alla teoria elementare dei numeri. Knuth [122] contiene una buona discussione sugli algoritmi per trovare il massimo comun divisore e anche altri importanti algoritmi nella teoria dei numeri. Riesel [168] e Bach [16] forniscono un compendio più recente della teoria computazionale dei numeri. Dixon [56] dà una panoramica della scomposizione in fattori e della verifica di primalità. Gli atti della conferenza pubblicati da Pomerance [159] contengono alcuni piacevoli articoli di rassegna.

Knuth [122] discute le origini dell'algoritmo di Euclide. Esso appare nel Libro 7, proposizioni 1 e 2, degli *Elementi* del matematico greco Euclide, scritto attorno al 300 a.C. La descrizione di Euclide può essere stata tratta da un algoritmo dovuto a Eudosso intorno al 375 a.C. L'algoritmo di Euclide può vantare l'onore di essere il più antico algoritmo non banale: suo unico rivale è l'algoritmo del contadino russo per la moltiplicazione (si veda il Capitolo 29) che era noto agli antichi egizi.

Knuth attribuisce un caso speciale del teorema del resto cinese (Teorema 33.27) al matematico cinese Sun-Tsù che visse nel periodo tra il 200 a.C e il 200 d.C. – la data è piuttosto incerta. Lo stesso caso speciale fu fornito dal matematico greco Nicomaco attorno al 100 d.C. Esso fu generalizzato da Chhin Chiu-Shao nel 1247. Il teorema cinese del resto fu formulato e dimostrato nella sua completa generalità da L. Eulero nel 1734.

L'algoritmo randomizzato di verifica della primalità è dovuto a Miller [147] e Rabin [166]: esso è l'algoritmo randomizzato di verifica della primalità più veloce conosciuto finora, a meno di un fattore costante. La dimostrazione del Teorema 33.39 è leggermente diversa da quella suggerita da Bach [15]. Una dimostrazione di un risultato più forte per MILLER-RABIN fu data da Monier [148, 149]. La randomizzazione sembra essere necessaria per ottenere un algoritmo di verifica della primalità con tempo polinomiale. L'algoritmo deterministico di verifica della primalità più veloce conosciuto finora è la versione Cohen-Lenstra [45] della verifica di primalità di Adleman, Pomerance e Rumely [3]. Quando si verifica la primalità di un numero n di lunghezza $\lceil \lg(n + 1) \rceil$, tale algoritmo richiede tempo $(\lg n)^{c \lg \lg \lg n}$ che è leggermente superpolinomiale.

Il problema di trovare numeri primi "casuali" grandi è discusso in modo interessante in un articolo di Beauchemin, Brassard, Crépeau, Goutier e Pomerance [20].

Il concetto di sistema di crittografia a chiave pubblica è dovuto a Diffie e Hellman [54]. Il sistema di crittografia RSA fu proposto nel 1977 da Rivest, Shamir e Adleman [169]. Il campo della crittografia è da allora fiorito. In particolare sono state sviluppate molte nuove tecniche per dimostrare la sicurezza dei sistemi di crittografia. Per esempio Goldwasser e Micali [86] mostrano che la randomizzazione può essere uno strumento efficace nel progetto di sistemi di crittografia a chiave pubblica sicuri. Per lo schema della firma, Goldwasser, Micali e Rivest [88] presentano uno schema di firme numeriche per cui ogni tipo di contraffazione concepibile è difficile quanto la scomposizione in fattori primi. Di recente Goldwasser, Micali e Rackoff [87] hanno introdotto una classe di schemi di crittografia a "conoscenza zero" per cui può essere dimostrato (sotto alcune ragionevoli ipotesi) che nessuna controparte apprende più di quanto si suppone possa apprendere da una comunicazione.

L'euristica rho per la scomposizione di interi in fattori è stata inventata da Pollard [156]. La versione presentata qui è una variante proposta da Brent [35].

Il miglior algoritmo per la scomposizione di numeri grandi ha un tempo di esecuzione che cresce esponenzialmente con la radice quadrata della lunghezza del numero n da fattorizzare. L'algoritmo di scomposizione in fattori a "setaccio-quadratico" dovuto a Pomerance [158], è forse in generale il più efficiente di tali algoritmi per grandi input. Sebbene sia difficile dare un'analisi rigorosa di questo algoritmo, sotto ragionevoli ipotesi si può derivare una stima di $L(n)^{1+o(1)}$ del tempo di esecuzione, dove $L(n) = e^{\sqrt{\ln n \ln\ln n}}$. Il metodo della curva ellittica dovuto a Lenstra [137] può essere più efficace per qualche input del metodo del setaccio quadratico, poiché, come il metodo rho di Pollard, esso può trovare un fattore primo piccolo p abbastanza rapidamente. Con questo metodo, il tempo per trovare p è stimato essere $L(p)^{\sqrt{2}+o(1)}$.

Corrispondenza tra stringhe

34

Trovare tutte le occorenze di una stringa all'interno di un testo è un problema che si presenta frequentemente nei programmi di scrittura di testi. Tipicamente, il testo è il documento che si sta scrivendo e la stringa cercata, che sarà chiamata anche ***pattern***, è una parola particolare fornita dall'utente. Algoritmi efficienti per questo problema possono aumentare fortemente la ricchezza del programma di scrittura di testi. Gli algoritmi di corrispondenza tra stringhe sono usati tra l'altro anche per cercare particolari sottosequenze nelle sequenze del DNA.

Si formalizza il problema della ***corrispondenza tra stringhe*** come segue. Si assume che il testo sia un array $T[1 .. n]$ di lunghezza n e che il pattern sia un array $P[1 .. m]$ di lunghezza $m \leq n$. Si suppone inoltre che gli elementi di P e T siano caratteri presi da un alfabeto finito Σ. Per esempio, si può avere $\Sigma = \{0,1\}$ oppure $\Sigma = \{a, b, \ldots, z\}$. Gli array di caratteri P e T sono spesso chiamati ***stringhe*** di caratteri.

Si dice che il pattern P ***appare con spostamento*** s nel testo T (o, equivalentemente, il pattern P ***occorre a cominciare dalla posizione*** $s + 1$ nel testo T) se $0 \leq s \leq n - m$ e $T[s + 1 .. s + m] = P[1 .. m]$ (cioè, se $T[s + j] = P[j]$, per $1 \leq j \leq m$). Se P appare con spostamento s in T, allora s si chiama ***spostamento valido***, altrimenti s si chiama ***spostamento non valido***. Il problema della corrispondenza tra stringhe è il problema di trovare tutti gli spostamenti validi con cui un dato pattern P appare in un dato testo T. La figura 34.1 illustra queste definizioni.

Questo capitolo è organizzato come segue. Nel paragrafo 34.1 si passa in rassegna un algoritmo ingenuo forza-bruta per risolvere il problema della corrispondenza tra stringhe, che ha tempo di esecuzione $O((n - m + 1)m)$ nel caso peggiore. Il paragrafo 34.2 presenta un interessante algoritmo per la corrispondenza tra stringhe, dovuto a Rabin e Karp. Anche questo algoritmo ha tempo di esecuzione $O((n - m + 1)m)$ nel caso peggiore, ma in pratica e nel caso medio funziona molto meglio. Inoltre l'algoritmo si generalizza facilmente ad altri problemi di corrispondenza tra stringhe. Il paragrafo 34.3 descrive quindi un algoritmo per la corrispondenza tra stringhe che comincia con la costruzione di un automa a stati finiti progettato specificamente per cercare le occorrenze di un dato pattern P in un testo. Questo algoritmo impiega tempo $O(n + m|\Sigma|)$. L'algoritmo di Knuth-Morris-Pratt, (o KMP), simile ma molto più chiaro, è presentato al paragrafo 34.4; l'algoritmo KMP impiega tempo $O(n + m)$. Infine, il paragrafo 34.5 descrive un algoritmo dovuto a Boyer e Moore che è spesso la migliore scelta pratica, sebbene il suo tempo di esecuzione nel caso peggiore (come quello dell'algoritmo Rabin-Karp) non sia migliore di quello dell'algoritmo ingenuo per la corrispondenza tra stringhe.

Figura 34.1 *Il problema della corrispondenza tra stringhe. L'obiettivo è di trovare tutte le occorrenze del pattern P = abaa nel testo T = abcabaabcabac. Il pattern appare solo una volta nel testo, con spostamento s = 3. Lo spostamento s = 3 è detto spostamento valido. Ogni carattere del pattern è connesso con una linea verticale al carattere corrispondente nel testo e tutti i caratteri che coincidono sono mostrati in grigio.*

Notazione e terminologia

Si denoterà con Σ^* (si legga "Σ stella") l'insieme di tutte le stringhe di lunghezza finita formate usando caratteri dell'alfabeto Σ. In questo capitolo si considerano solo stringhe di lunghezza finita. Anche la ***stringa vuota*** di lunghezza zero, denotata con ε, appartiene a Σ^*. La lunghezza di una stringa è denotata con $|x|$. La ***concatenazione*** di due stinghe x e y, denotata con xy, ha lunghezza $|x| + |y|$ e consiste dei caratteri di x seguiti dai caratteri di y.

Si dice che una stringa w è un ***prefisso*** di una stringa x, e si denota con $w \sqsubset x$, se $x = wy$ per qualche stringa $y \in \Sigma^*$. Si noti che se $w \sqsubset x$, allora $|w| \leq |x|$. Analogamente, si dice che una stringa w è un ***suffisso*** di una stringa x, e si denota con $w \sqsupset x$, se $x = yw$ per qualche $y \in \Sigma^*$. Da $w \sqsupset x$ segue che $|w| \leq |x|$. La stringa vuota ε è sia prefisso che suffisso di ogni stringa. Per esempio si ha ab $\sqsubset$ abcca e cca $\sqsupset$ abcca. È utile notare che per qualunque stringa x e y e qualunque carattere a, si ha $x \sqsupset y$ se e solo se $ax \sqsupset ay$. Si noti inoltre che $\sqsubset$ e $\sqsupset$ sono relazioni transitive. In seguito sarà utile il seguente lemma.

Lemma 34.1 (Lemma di sovrapposizione dei suffissi)
Si supponga che x, y e z siano stringhe tali che $x \sqsupset z$ e $y \sqsupset z$. Se $|x| \leq |y|$, allora $x \sqsupset y$. Se $|x| \geq |y|$, allora $y \sqsupset x$. Se $|x| = |y|$, allora $x = y$.

Dimostrazione. Si osservi la figura 34.2 per una dimostrazione grafica ■

Per brevità di notazione, si denoterà il prefisso di k caratteri $P[1..k]$ del pattern $P[1..m]$ con P_k. Pertanto, $P_0 = \varepsilon$ e $P_m = P = P[1..m]$. Analogamente si denota il prefisso di k caratteri del testo T con T_k. Usando questa notazione, si può dire che il problema della corrispondenza tra stringhe è quello di trovare tutti gli spostamenti s nell'intervallo $0 \leq s \leq n - m$ tali che $P \sqsupset T_{s+m}$.

Nello pseudocodice, il confronto tra due stringhe di uguale lunghezza si considera come un'operazione primitiva. Se le stringhe sono confrontate da sinistra a destra e il confronto si ferma quando è scoperta una discordanza, si assume che il tempo richiesto da tale controllo sia una funzione lineare del numero di caratteri corrispondenti scoperti. Per essere precisi, si assume che il controllo "$x = y$" richieda tempo $\Theta(t + 1)$, dove t è la lunghezza della stringa z più lunga tale che $z \sqsubset x$ e $z \sqsubset y$.

34.1 Algoritmo ingenuo di corrispondenza tra stringhe

L'algoritmo ingenuo trova tutti gli spostamenti validi usando un ciclo che controlla la condizione $P[1..m] = T[s + 1..s + m]$ per ognuno degli $n - m + 1$ possibili valori di s.

Figura 34.2 *Una dimostrazione grafica del lemma 34.1. Si suppone che $x \sqsupset z$ e $y \sqsupset z$. Le tre parti della figura illustrano i tre casi del lemma. Le linee verticali connettono porzioni corrispondenti (mostrate in grigio) delle stringhe.* **(a)** *Se $|x| \leq |y|$, allora $x \sqsupset y$.* **(b)** *Se $|x| \geq |y|$, allora $y \sqsupset x$.* **(c)** *Se $|x| = |y|$, allora $x = y$.*

```
NAIVE-STRING-MATCHER(T,P)
1  n ← length[T]
2  m ← length[P]
3  for s ← 0 to n–m
4          do if  P [1 .. m] = T[s + 1 .. s + m]
5                  then stampa "Il pattern appare con spostamento" s
```

La procedura ingenua di corrispondenza tra stringhe può essere interpretata graficamente come lo scorrimento sul testo di una "configurazione" contenente il pattern, rilevando gli spostamenti in cui tutti i caratteri della configurazione sono uguali ai corrispondenti caratteri sul testo, come illustrato nella figura 34.3. Il ciclo **for** che comincia alla linea 3 considera esplicitamente ogni possibile spostamento. Il controllo alla linea 4 determina se lo spostamento corrente è valido; questo controllo prevede un ciclo implicito per verificare le posizioni dei caratteri finché o tutti i caratteri combaciano oppure è trovata una discordanza. La linea 5 stampa ogni spostamento valido s.

La procedura NAIVE-STRING-MATCHER richiede tempo $\Theta((n - m + 1)m)$ nel caso peggiore. Per esempio, si consideri il testo a^n (una stringa di n a) e il pattern a^m. Per convalidare ognuno degli $n - m + 1$ possibili valori dello spostamento s, il ciclo implicito della linea 4 che confronta caratteri corrispondenti deve essere eseguito m volte per convalidare lo spostamento. Pertanto il tempo di esecuzione nel caso peggiore è $\Theta((n - m + 1)m)$, che è $\Theta(n^2)$ se $m = \lfloor n/2 \rfloor$.

Come si vedrà, NAIVE-STRING-MATCHER non è una procedura ottima per questo problema. In effetti, in questo capitolo si mostrerà un algoritmo con tempo di esecuzione $O(n + m)$ nel caso peggiore. L'algoritmo ingenuo di corrispondenza tra stringhe è inefficiente perchè le informazioni ottenute sul testo per un certo valore di s sono completamente ignorate quando si considerano altri valori di s. Tali informazioni possono invece essere molto utili. Per esempio se P = aaab e si trova che $s = 0$ è valido, allora nessuno degli spostamenti 1, 2 o 3 è valido, poiché $T[4]$ = b. Nei seguenti paragrafi, si esaminano alcuni modi per utilizzare in modo efficace questo tipo di informazione.

Figura 34.3 *Il funzionamento dell'algoritmo ingenuo di corrispondenza tra stringhe per il pattern* P = aab *e il testo* T = acaabc. *Si può immaginare il pattern P come una "configurazione" che viene fatta via via scorrere. Le parti* (a)-(d) *mostrano i quattro allineamenti successivi tentati dall'algoritmo. In ogni parte, le linee verticali connettono le porzioni che combaciano (mostrate in grigio), e una linea spezzata connette, se esiste, il primo carattere discordante. Un'occorrenza del pattern viene trovata, con spostamento* $s = 2$, *come mostrato nella parte* (c).

Esercizi

34.1-1 Si mostrino i confronti eseguiti dall'algoritmo ingenuo di corrispondenza tra stringhe per il pattern P = 0001 nel testo T = 000010001010001.

34.1-2 Si mostri che il tempo dell'algoritmo ingenuo di corrispondenza tra stringhe per trovare la *prima* occorrenza di un pattern in un testo è $\Theta((n-m+1)(m-1))$ nel caso peggiore.

34.1-3 Si supponga che tutti i caratteri nel pattern P siano diversi. Si mostri come si può rendere più efficiente NAIVE-STRING-MATCHER in modo che richieda tempo $O(n)$ su un testo T di n caratteri.

34.1-4 Si supponga che il pattern P e il testo T siano stringhe di lunghezza m ed n rispettivamente, scelte *casualmente* dall'alfabeto d-ario $\Sigma_d = \{0, 1, \ldots, d-1\}$ dove $d \geq 2$. Si mostri che il numero *medio* di confronti carattere per carattere eseguiti dal ciclo implicito alla linea 4 dell'algoritmo ingenuo è

$$(n-m+1)\frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1) .$$

(Si assuma che l'algoritmo ingenuo termini il confronto dei caratteri per un dato spostamento quando trova una discordanza o quando l'intero pattern coincide.) Pertanto, per stringhe scelte casualmente, l'algoritmo ingenuo è abbastanza efficiente.

34.1-5 Si supponga di consentire che il pattern P contenga occorrenze di un *carattere jolly* ◊ che corrisponde a una stringa *arbitraria* di caratteri (anche di lunghezza zero). Per esempio, il pattern ab◊ba◊c appare nel testo cabccbacbacab come

c ab cc ba cba c ab
ab ◊ ba ◊ c

e come

c ab ccbac ba c ab
ab ◊ ba ◊ c

Si noti che il carattere jolly può ripetersi un numero arbitrario di volte nel pattern ma si suppone che non compaia affatto nel testo. Si dia un algoritmo di tempo polinomiale per determinare se un tale pattern P appare in un dato testo T e si analizzi il tempo di esecuzione dell'algoritmo.

34.2 Algoritmo Rabin-Karp

Rabin e Karp hanno proposto un algoritmo per la corrispondenza tra stringhe che si comporta bene in pratica e che inoltre si può generalizzare ad altri algoritmi per problemi correlati, come ad esempio la corrispondenza bidimensionale tra stringhe. Il tempo di esecuzione dell'algoritmo Rabin-Karp è $O((n-m+1)m)$ nel caso peggiore, ma ha un buon tempo di esecuzione nel caso medio.

Quest'algoritmo fa uso di nozioni elementari di teoria dei numeri come l'equivalenza di due numeri modulo un terzo numero. Si può fare riferimento al paragrafo 33.1 per le definizioni principali.

Si assuma, per chiarezza, che $\Sigma = \{0, 1, 2, \ldots, 9\}$, cioè ogni carattere sia una cifra decimale. (Nel caso generale, si può assumere che ogni carattere sia una cifra in notazione base d, dove $d = |\Sigma|$.) Si può allora vedere una stringa di k caratteri consecutivi come un numero decimale di lunghezza k. La stringa di caratteri 31415 pertanto corrisponde al numero decimale 31415. Data la duplice interpretazione dei caratteri in input sia come simboli grafici che come cifre, è conveniente in questo paragrafo denotarli come cifre nella fonte di testo standard.

Dato il pattern $P[1 .. m]$, si denoti con p il suo corrispondente valore decimale. Analogamente, dato un testo $T[1 .. n]$, si denoti con t_s il valore decimale della sottostringa $T[s+1 .. s+m]$ di lunghezza m, per $s = 0, 1, \ldots, n-m$. Certamente, $t_s = p$ se e solo se $T[s+1 .. s+m] = P[1 .. m]$: pertanto, s è uno spostamento valido se e solo se $t_s = p$. Se si potesse calcolare p in tempo $O(m)$ e tutti i valori t_i in un tempo totale $O(n)$, allora si potrebbero determinare tutti gli spostamenti validi s in tempo $O(n)$ confrontando p con ogni t_s. (Per il momento non ci si preoccupi della possibilità che p e t_s possano essere numeri molto grandi.)

Si può calcolare p in tempo $O(m)$ usando la regola di Horner (si veda il paragrafo 32.1):

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10P[1])\ldots)) .$$

Il valore t_0 può essere analogamente calcolato a partire da $T[1 .. m]$ in tempo $O(m)$.

Per calcolare i rimanenti valori $t_1, t_2, \ldots, t_{n-m}$ in tempo $O(n-m)$, è sufficiente osservare che t_{s+1} può essere calcolato da t_s in tempo costante, poiché

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] . \tag{34.1}$$

Per esempio, se $m = 5$ e $t_s = 31415$, allora si desidera rimuovere la cifra più significativa $T[s+1] = 3$ e inserire una nuova cifra meno significativa (si supponga sia $T[s+5+1] = 2$) per ottenere

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152 . \end{aligned}$$

Sottraendo $10^{m-1}T[s+1]$ si rimuove la cifra più significativa da t_s, moltiplicando il risultato per 10 si trasla il numero di una posizione a sinistra e addizionando $T[s+m+1]$ si inserisce l'appropriata cifra meno significativa. Se la costante 10^{m-1} è calcolata prima (il che può essere fatto in tempo $O(\lg m)$ usando le tecniche al paragrafo 33.6, sebbene per quest'applicazione sia abbastanza adeguato anche un metodo diretto $O(m)$), allora ogni esecuzione dell'equazio-

ne (34.1) richiede un numero costante di operazioni aritmetiche. Perciò, p e $t_0, t_1, \ldots, t_{n-m}$ possono essere calcolati tutti in tempo $O(n + m)$ e si possono trovare tutte le occorrenze del pattern $P[1 .. m]$ nel testo $T[1 .. n]$ in tempo $O(n + m)$.

L'unica difficoltà di questa procedura è che p e t_s possono essere troppo grandi per funzionare in modo conveniente. Se P contiene m caratteri, allora l'ipotesi che ogni operazione aritmetica su p (che è lungo m cifre) richieda tempo costante è irragionevole. Fortunatamente, vi è un semplice metodo per risolvere questo problema, come mostrato in figura 34.4: si calcolano p e i t_s modulo un appropriato numero q. Poiché il calcolo di p, t_0 e la ricorrenza (34.1) possono essere eseguiti tutti modulo q, si vede che p e tutti i t_s possono essere calcolati modulo q in tempo $O(n + m)$. Tipicamente come modulo q è scelto un primo tale che $10q$ sia rappresentabile in una parola della macchina, il che consente di eseguire tutti i calcoli necessari con precisione aritmetica singola. In generale, con un alfabeto d-ario $\{0, 1, \ldots, d - 1\}$, si sceglie q tale che dq sia rappresentabile in una parola della macchina e si adatta l'equazione (34.1) della ricorrenza perchè funzioni modulo q, così che diventi

$$t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q , \qquad (34.2)$$

dove $h \equiv d^{m-1} \pmod q$ è il valore della cifra "1" nella posizione più significativa di una porzione di testo di m cifre.

La soluzione di lavorare modulo q però presenta un piccolo inconveniente, poiché $t_s \equiv p \pmod q$ non implica $t_s = p$. D'altra parte se $t_s \not\equiv p \pmod q$, allora sicuramente $t_s \neq p$, quindi lo spostamento s non è valido. Si può pertanto usare il controllo $t_s \equiv p \pmod q$ come veloce controllo euristico per scartare gli spostamenti s non validi. Qualunque spostamento s per cui $t_s \equiv p \pmod q$ deve essere ulteriormente controllato per vedere se s è realmente valido o è soltanto un ***colpo mancato***. Questo test può essere fatto esplicitamente controllando la condizione $P[1 .. m] = T[s + 1 .. s + m]$. Se q è sufficientemente grande, allora si può sperare che i colpi mancati si verifichino così poco di frequente che il costo dell'ulteriore controllo sia basso.

La seguente procedura precisa questi concetti. Gli input della procedura sono il testo T, il pattern P, la base d da usare (che tipicamente è $|\Sigma|$) e il primo q da usare.

```
RABIN-KARP-MATCHER(T,P,d,q)
1  n ← length[T]
2  m ← length[P]
3  h ← d^(m-1) mod q
4  p ← 0
5  t0 ← 0
6  for i ← 0 to m
7      do p ← (dp + P[i]) mod q
8         t0 ← (dt0 + T[i]) mod q
9  for s ← 0 to n-m
10     do if p = ts
11           then if P[1..m] = T[s+1..s+m]
12                then "Il pattern appare con spostamento" s
13        if s < n - m
14           then ts+1 = (d(ts - T[s+1]h) + T[s+m+1]) mod q
```

Figura 34.4 *L'algoritmo Rabin-Karp. Ogni carattere è una cifra decimale e si eseguono i calcoli modulo 13.* (a) *Un testo. Una porzione di lunghezza 5 è mostrata in grigio. Il valore numerico del numero in grigio è calcolato modulo 13, ottenendo il valore 7.* (b) *Lo stesso testo con i valori calcolati modulo 13 per ogni possibile posizione di una porzione di lunghezza 5. Assumendo il pattern P = 31415, si cercano le finestre il cui valore modulo 13 sia 7, poiché 31415 ≡ 7 (mod 13). Vengono trovate due di tali finestre, evidenziate in grigio nella figura. La prima, che comincia alla posizione 7 del testo, è in effetti un'occorrenza del pattern, mentre la seconda, che comincia alla posizione 13 del testo, è un colpo mancato.* (c) *Il calcolo del valore di una finestra in tempo costante, dato il valore della finestra precedente. La prima finestra ha valore 31415. Sottraendo la cifra più significativa 3, traslando a sinistra (moltiplicando per 10), quindi sommando 2 alla cifra meno significativa, si ottiene il nuovo valore 14152. I calcoli sono eseguiti modulo 13, così il valore della prima finestra è 7 e il valore calcolato per la nuova finestra è 8.*

La procedura RABIN-KARP-MATCHER funziona come segue. Tutti i caratteri sono interpretati come cifre in base d. Gli indici di t sono forniti solo per chiarezza; il programma funziona correttamente se tutti gli indici sono ignorati. Alla linea 3 si inizializza h al valore della cifra più significativa di una porzione di m cifre. Le linee 4-8 calcolano p come il valore di $P[1 .. m] \bmod q$ e t_0 come il valore di $T[1 .. m] \bmod q$. Il ciclo **for** che comincia alla linea 9 si ripete per tutti i possibili spostamenti s. Il ciclo presenta la seguente invariante: quando la linea 10 viene eseguita, $t_s = T[s + 1 .. s + m] \bmod q$. Se $p = t_s$ alla linea 10 (un "successo"), allora alla linea

11 si controlla se $P[1..m] = T[s+1..s+m]$ per escludere la possibilità di un colpo mancato. Qualunque spostamento valido venga trovato è stampato alla linea 12. Se $s < n - m$ (verificato nella linea 13), allora il ciclo **for** deve essere eseguito almeno un'altra volta e così la linea 14 viene eseguita per assicurare che l'invariante del ciclo continui a valere quando sarà raggiunta di nuovo la linea 10. Alla linea 14 si calcola il valore di t_{s+1} mod q dal valore di t_s mod q in tempo costante usando direttamente l'equazione (34.2).

Il tempo di esecuzione di RABIN-KARP-MATCHER è $\Theta((n-m+1)m)$ nel caso peggiore, poiché (come l'algoritmo ingenuo di corrispondenza tra stringhe) l'algoritmo Rabin-Karp verifica esplicitamente ogni spostamento valido. Se $P = a^m$ e $T = a^n$, allora le verifiche richiedono tempo $\Theta((n-m+1)m)$, dato che ognuno degli $n-m+1$ possibili spostamenti è valido. (Si noti anche che il calcolo di d^{m-1} mod q alla linea 3 e il ciclo alle linee 6-8 richiedono tempo $O(m) = O((n-m+1)m)$.)

In molte applicazioni, ci si aspettano pochi spostamenti validi (probabilmente $O(1)$) e così il tempo di esecuzione atteso dell'algoritmo è $O(n+m)$ più il tempo richiesto per la verifica dei colpi mancati. Si può basare un'analisi euristica sull'ipotesi che la riduzione ai valori modulo q si comporta come una corrispondenza casuale tra Σ^* e Z_q. (Si veda la discussione sull'uso della divisione per definire funzioni hash al paragrafo 12.3.1. È difficile formalizzare e dimostrare l'ipotesi precedente, sebbene un approccio ammissibile sia di assumere che q venga scelto in modo casuale tra gli interi di dimensione appropriata. Qui si non approfondirà ulteriormente l'argomento.) Quindi, ci si può aspettare che il numero di colpi mancati sia $O(n/q)$, poiché la possibilità che un t_s arbitrario sia equivalente a p modulo q, può essere stimata come $1/q$. Il tempo medio richiesto dall'algoritmo Rabin-Karp è allora

$$O(n) + O(m(v + n/q)),$$

dove v è il numero di spostamenti validi. Questo tempo di esecuzione è $O(n)$ se si sceglie $q \geq m$. Cioè, se il numero medio di spostamenti validi è piccolo ($O(1)$) e il primo q è scelto più grande della lunghezza del pattern, allora ci si può aspettare che la procedura Rabin-Karp venga eseguita in tempo $O(n+m)$.

Esercizi

34.2-1 Lavorando modulo q, quanti colpi mancati incontra l'algoritmo Rabin-Karp nel testo $T = 3141592653589793$ quando cerca il pattern $P = 26$?

34.2-2 Come si può estendere il metodo di Rabin-Karp al problema di cercare in un testo un'occorrenza di un qualunque pattern da un dato insieme di k pattern?

34.2-3 Si mostri come estendere il metodo di Rabin-Karp per gestire il problema della ricerca di un dato pattern $m \times m$ in un array $n \times n$ di caratteri. (Il pattern potrebbe essere spostato verticalmente e orizzontalmente, ma non può essere ruotato.)

34.2-4 Alice ha una copia di un file $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$ lungo n bit e Bob analogamente ha un file $B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$ lungo n bit. Alice e Bob desiderano conoscere se i loro file sono identici. Per evitare la trasmissione degli interi file A e B, essi usano il seguente controllo probabilistico veloce. Insieme selezionano un numero primo $q > 1000n$ e casualmente selezionano un intero x da $\{0, 1, \ldots, q-1\}$. Allora, Alice valuta

$$A(x) = \left(\sum_{i=0}^{n} a_i x^i \right) \bmod q$$

e Bob valuta $B(x)$ in modo analogo. Si dimostri che se $A \neq B$, vi è al più una possibilità su 1000 che $A(x) = B(x)$, mentre se i due file sono uguali, $A(x)$ è necessariamente uguale a $B(x)$. (*Suggerimento:* si veda l'Esercizio 33.4-4.)

34.3 Corrispondenza tra stringhe con gli automi a stati finiti

Molti algoritmi per la corrispondenza tra stringhe costruiscono un automa a stati finiti che scandisce il testo T alla ricerca di tutte le occorrenze del pattern P. Questo paragrafo presenta un metodo per costruire un tale automa. Questi automi di corrispondenza tra stringhe sono molto efficienti: esaminano ogni carattere del testo *esattamente una volta*, richiedendo un tempo costante per ogni carattere del testo. Il tempo impiegato – dopo che l'automa è stato costruito – è quindi $\Theta(n)$. Il tempo per costruire l'automa, tuttavia, può essere grande se Σ è grande. Il paragrafo 34.4 descrive un ingegnoso metodo relativo a questo problema.

Si comincia questo paragrafo con la definizione di automa a stati finiti. Quindi si esamina un automa particolare di corrispondenza tra stringhe e si mostra come questo possa essere usato per trovare le occorrenze di un pattern in un testo. Questa discussione include dettagli su come simulare il comportamento di un automa di corrispondenza tra stringhe su un dato testo. Infine si mostrerà come costruire un automa di corrispondenza tra stringhe per un dato pattern di input.

Automi a stati finiti

Un *automa a stati finiti* M è una quintupla $(Q, q_0, A, \Sigma, \delta)$, dove

- Q è un insieme finito di *stati*,
- $q_0 \in Q$ è lo *stato iniziale*,
- $A \subseteq Q$ è un insieme di *stati accettanti*,
- Σ è un *alfabeto di input* finito,
- δ è una funzione da $Q \times \Sigma$ in Q, chiamata *funzione di transizione* di M.

L'automa a stati finiti comincia dallo stato q_0 e legge i caratteri della stringa di input uno alla volta. Se l'automa è nello stato q e legge il carattere in input a, si muove ("fa una transizione") dallo stato q allo stato $\delta(q, a)$. Quando il suo stato corrente q appartiene ad A, si dice che la macchina M *accetta* la stringa appena letta. Un input che non è accettato è detto *rifiutato*. La figura 34.5 illustra queste definizioni con un semplice automa a due stati.

Un automa a stati finiti M induce una funzione ϕ, chiamata *funzione stato-finale*, da Σ^* a Q tale che $\phi(w)$ è lo stato in cui M si trova al termine della scansione della stringa w.

Figura 34.5 *Un semplice automa a stati finiti con insieme di stati $Q = \{0, 1\}$, stato iniziale $q_0 = 0$ e alfabeto di input $\Sigma = \{a, b\}$.* (a) *Una rappresentazione tabulare della funzione di transizione δ.* (b) *Un diagramma di transizione degli stati equivalente. Lo stato 1 è l'unico stato accettante (evidenziato in nero). Gli archi orientati rappresentano le transizioni. Per esempio, l'arco dallo stato 1 allo stato 0 etichettato con* b *indica $\delta(1, b) = 0$. Questo automa accetta quelle stringhe che terminano con un numero dispari di* a. *Più precisamente, una stringa x è accettata se e solo se $x = yz$, dove $y = \varepsilon$ o y termina con una* b *e $z = a^k$, dove k è dispari. Per esempio, la sequenza di stati che quest'automa attraversa su input* abaaa *(incluso lo stato iniziale) è $\langle 0, 1, 0, 1, 0, 1\rangle$; dunque, tale input è accettato. Per l'input* abbaa, *la sequenza di stati è $\langle 0, 1, 0, 0, 1, 0\rangle$; quindi tale input non è accettato.*

Pertanto, M accetta una stringa w se e solo se $\phi(w) \in A$. La funzione ϕ è definita dalla relazione ricorsiva

$$\phi(\varepsilon) = q_0 ,$$
$$\phi(wa) = \delta(\phi(w), a) \qquad \text{per } w \in \Sigma^*, a \in \Sigma.$$

Automi per la corrispondenza tra stringhe

Per ogni pattern P vi è un automa di corrispondenza tra stringhe; quest'automa deve essere costruito a partire dal pattern in una fase di preelaborazione prima di poter essere usato per eseguire la ricerca nel testo. La figura 34.6 illustra questa costruzione per il pattern $P =$ ababaca. D'ora in poi, si assumerà che P sia una data stringa pattern: per brevità non si indicherà la dipendenza da P nella notazione.

Per specificare l'automa di corrispondenza tra stringhe relativo a un dato pattern $P[1 .. m]$, si definisce prima una funzione σ ausiliaria, chiamata ***funzione suffisso*** relativa a P. La funzione σ è una corrispondenza tra Σ^* e $\{0, 1, \ldots, m\}$ tale che $\sigma(x)$ è la lunghezza del più lungo prefisso di P che è un suffisso di x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\} .$$

La funzione suffisso σ è ben definita poiché la stringa vuota $P_0 = \varepsilon$ è un suffisso di ogni stringa. Per esempio, per il pattern $P =$ ab, si ha $\sigma(\varepsilon) = 0$, $\sigma($ccaca$) = 1$ e $\sigma($ccab$) = 2$. Per un pattern P di lunghezza m, si ha $\sigma(x) = m$ se e solo se $P \sqsupset x$. Segue dalla definizione della funzione suffisso che se $x \sqsupset y$, allora $\sigma(x) \le \sigma(y)$.

Si definisce l'automa di corrispondenza tra stringhe relativo a un dato pattern $P[1 .. m]$ come segue.

- L'insieme di stati Q è $\{0, 1, \ldots, m\}$. Lo stato iniziale q_0 è lo stato 0 e lo stato m è l'unico stato accettante.
- La funzione di transizione δ è definita dalla seguente equazione, per qualunque stato q e qualunque carattere a:

$$\delta(q, a) = \sigma(P_q a) . \tag{34.3}$$

Figura 34.6 (a) *Un diagramma di transizione degli stati per l'automa di corrispondenza tra stringhe che accetta tutte le stringhe terminanti con la stringa* ababaca. *Lo stato 0 è lo stato iniziale e lo stato 7 (evidenziato in nero) è l'unico stato accettante. Un arco orientato dallo stato i allo stato j etichettato con a rappresenta $\delta(i, a) = j$. Gli archi orientati verso destra che formano la "spina" dell'automa, evidenziati nella figura in grassetto, sono relativi a corrispondenze con successo tra il pattern e i caratteri in input. Gli archi orientati verso sinistra sono relativi a corrispondenze fallite. Alcuni archi che sono relativi a corrispondenze fallite non sono mostrati; per convenzione, se uno stato i non ha archi di uscita etichettati con a per qualche $a \in \Sigma$, allora $\delta(i, a) = 0$.* (b) *La funzione di transizione δ e la stringa pattern $P =$* ababaca. *In grigio sono mostrate le caselle della tabella relative a corrispondenze con successo tra il pattern e i caratteri in input.* (c) *Il comportamento dell'automa sul testo $T =$* ababababacaba. *Sotto ogni carattere del testo $T[i]$ è dato lo stato $\phi(T_i)$ in cui si trova l'automa dopo aver elaborato il prefisso T_i. Viene trovata un'occorrenza del pattern che termina nella posizione 9.*

La base logica intuitiva per la definizione di $\delta(q, a) = \sigma(P_q a)$ è la seguente. La macchina mantiene come invariante del suo comportamento che

$$\phi(T_i) = \sigma(T_i) ; \tag{34.4}$$

questo risultato è dimostrato in seguito nel Teorema 34.4. In parole povere, significa che dopo aver scandito i primi i caratteri del testo T, la macchina è nello stato $\phi(T_i) = q$, dove $q = \sigma(T_i)$ è la lunghezza del suffisso più lungo di T_i che sia anche un prefisso del pattern P. Se il prossimo carattere esaminato è $T[i + 1] = a$, allora la macchina dovrebbe effettuare una transizione nello stato $\sigma(T_{i+1}) = \sigma(T_i a)$. La dimostrazione del teorema mostra che $\sigma(T_i a) = \sigma(P_q a)$. Cioè, per calcolare la lunghezza del suffisso più lungo di $T_i a$ che è un prefisso di P, si può calcolare il più lungo suffisso di $P_q a$ che è un prefisso di P. In ogni stato, la macchina ha bisogno solo di conoscere la lunghezza del più lungo prefisso di P che è un suffisso di quanto è stato letto finora. Quindi, ponendo $\delta(q, a) = \sigma(P_q a)$ si mantiene l'invariante desiderata (34.4). Questa argomentazione informale sarà resa più rigorosa tra breve.

Nell'automa di corrispondenza tra stringhe della figura 34.6, per esempio, si ha $\delta(5, b) = 4$. Ciò segue dal fatto che se l'automa legge una b nello stato $q = 5$, allora $P_q b$ = ababab è il più lungo prefisso di P che è anche un suffisso di ababab è P_4 = abab.

Per chiarire il funzionamento di un automa di corrispondenza tra stringhe, si dà ora un programma semplice ed efficiente che simula il comportamento di un tale automa (rappresentato dalla sua funzione di transizione δ) nella ricerca delle occorrenze di un pattern P di lunghezza m in un testo di input $T[1 .. n]$. Come per qualsiasi automa di corrispondenza tra stringhe relativo a un pattern di lunghezza m, l'insieme di stati Q è $\{0, 1, \ldots, m\}$, lo stato iniziale è 0 e l'unico stato accettante è lo stato m.

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n ← length[T]
q ← 0
for i ← 1 to n
    do q ← δ(q, T[i])
       if q = m
          then s ← i-m
               stampa "Il pattern appare con spostamento" s
```

La struttura di FINITE-AUTOMATON-MATCHER con un semplice ciclo implica che il suo tempo di esecuzione su un testo di lunghezza n è $O(n)$. Questo tempo di esecuzione, però, non include il tempo richiesto per calcolare la funzione di transizione δ. Si affronterà questo problema in seguito, dopo aver dimostrato che la procedura FINITE-AUTOMATON-MATCHER funziona correttamente.

Si consideri il funzionamento dell'automa su un testo di input $T[1 .. n]$. Si dimostrerà che l'automa si trova nello stato $\sigma(T_i)$ dopo aver esaminato il carattere $T[i]$. Poiché $\sigma(T_i) = m$ se e solo se $P \sqsupset T_i$, la macchina è nello stato accettante m se e solo se il pattern P è stato appena scandito. Per dimostrare questo risultato, si fa uso dei due seguenti lemmi relativi alla funzione suffisso σ.

Lemma 34.2 (Diseguaglianza della funzione suffisso)

Per qualunque stringa x e qualunque carattere a, si ha $\sigma(xa) \le \sigma(x) + 1$.

Dimostrazione. Con riferimento alla figura 34.7, sia $r = \sigma(xa)$. Se $r = 0$ allora la conclusione $r \le \sigma(x) + 1$ è banalmente soddisfatta, poiché $\sigma(x)$ è sempre positiva. Per cui si assuma $r > 0$. Ora, $P_r \sqsupset xa$, dalla definizione di σ. Pertanto, $P_{r-1} \sqsupset x$, ignorando la a alla fine di P_r e alla fine di xa. Allora $r - 1 \le \sigma(x)$, poiché $\sigma(x)$ è il più grande k tale che $P_k \sqsupset x$. ■

Figura 34.7 *Un'illustrazione della dimostrazione del lemma 34.2. La figura mostra che $r \le \sigma(x) + 1$, dove $r = \sigma(xa)$.*

Figura 34.8 *Un'illustrazione della dimostrazione del lemma 34.3. La figura mostra che $r = \sigma(P_q a)$, dove $q = \sigma(x)$ e $r = \sigma(xa)$.*

Lemma 34.3 (Lemma di ricorsione della funzione suffisso)

Per qualunque stringa x e qualunque carattere a, se $q = \sigma(x)$, allora $\sigma(xa) \le \sigma(P_q a)$.

Dimostrazione. Dalla definizione di σ, si ha $P_q \sqsupset x$. Come è mostrato nella figura 34.8, $P_q a \sqsupset xa$. Se si pone $r = \sigma(xa)$, allora $r \le q + 1$ per il Lemma 34.2. Poiché $P_q a \sqsupset xa$, $P_r \sqsupset xa$ e $|P_r| \le |P_q a|$, il Lemma 34.1 implica che $P_r \sqsupset P_q a$. Quindi $r \le \sigma(P_q a)$, cioè $\sigma(xa) \le \sigma(P_q a)$. Ma si ha anche $\sigma(P_q a) \le \sigma(xa)$, poiché $P_q \sqsupset xa$. Pertanto $\sigma(xa) = \sigma(P_q a)$. ■

È possibile ora dimostrare il principale teorema che caratterizza il comportamento di un automa di corrispondenza tra stringhe su un dato testo in input. Come notato precedentemente, questo teorema mostra che l'automa tiene semplicemente traccia, ad ogni passo, del più lungo prefisso del pattern che è un suffisso di quanto è stato letto finora.

Teorema 34.4

Se ϕ è la funzione stato-finale di un automa di corrispondenza tra stringhe per un dato pattern P e $T[1 .. n]$ è un testo di input per l'automa, allora

$\phi(T_i) = \sigma(T_i)$

per $i = 0, 1, \ldots, n$.

Dimostrazione. La dimostrazione è per induzione su i. Per $i = 0$, il teorema è banalmente vero, perché $T_0 = \varepsilon$. Pertanto, $\phi(T_0) = \sigma(T_0) = 0$.

Ora si assuma che $\phi(T_i) = \sigma(T_i)$ e si dimostri che $\phi(T_{i+1}) = \sigma(T_{i+1})$. Si denotino $\phi(T_i)$ con q e $T[i + 1]$ con a. Allora,

$$
\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(per definizione di } T_{i+1} \text{ e } a) \\
&= \delta(\phi(T_i), a) && \text{(per definizione di } \phi) \\
&= \delta(q, a) && \text{(per definizione di } q) \\
&= \sigma(P_q a) && \text{(per la definizione (34.3) di } \delta) \\
&= \sigma(T_i a) && \text{(per il lemma 34.3 e dall'induzione)} \\
&= \sigma(T_{i+1}) && \text{(per definizione di } T_{i+1})
\end{aligned}
$$

Per induzione il teorema è dimostrato. ■

Dal Teorema 34.4, se la macchina assume lo stato q alla linea 4, allora q è il valore più grande per cui $P_q \sqsupset T_i$. Quindi si ha $q = m$ alla linea 5 se e solo se è stata appena esaminata un'occorrenza del pattern P. Si può concludere che FINITE-AUTOMATON-MATCHER funziona correttamente.

Calcolo della funzione di transizione

La seguente procedura calcola la funzione di transizione δ per un dato pattern $P[1..m]$.

```
COMPUTE-TRANSITION-FUNCTION(P,Σ)
1  m ← length[P]
2  for q ← 0 to m
3      do for ogni carattere a ∈ Σ
4          do k ← min(m + 1, q + 2)
5              repeat k ← k − 1
6              until P_k ⊐ P_q a
7              δ(q,a) ← k
8  return δ
```

Questa procedura calcola $\delta(q,a)$ in modo diretto in base alla sua definizione. I cicli interni che cominciano alle linee 2 e 3 considerano tutti gli stati q e i caratteri a, mentre le linee 4-7 assegnano a $\delta(q, a)$ il più grande k tale che $P_k \sqsupset P_q a$. Il codice comincia con il valore di k più grande possibile che è $\min(m, q + 1)$ e decrementa k finché $P_k \sqsupset P_q a$.

Il tempo di esecuzione di COMPUTE-TRANSITION-FUNCTION è $O(m^3|\Sigma|)$, perché il ciclo più esterno contribuisce con un fattore $m|\Sigma|$, il ciclo più interno, **repeat**, può ripetersi al più $m + 1$ volte e la verifica $P_k \sqsupset P_q a$ alla linea 6 può richiedere fino a m confronti tra caratteri. Esistono procedure molto più veloci; il tempo richiesto per calcolare δ può essere abbassato a $O(m|\Sigma|)$ utilizzando alcune informazioni calcolate astutamente sul pattern P (si veda l'Esercizio 34.4-6). Con questa procedura migliorata per calcolare δ, il tempo di esecuzione totale per trovare tutte le occorrenze di un pattern di lunghezza m in un testo di lunghezza n su un alfabeto Σ è $O(n + m|\Sigma|)$.

Esercizi

34.3-1 Si costruisca l'automa di corrispondenza tra stringhe per il pattern P = `aabab`c si illustri il suo funzionamento sulla stringa testo:
T = `aaababaabaababaab`.

34.3-2 Si disegni un diagramma di transizione di stati per un automa di corrispondenza tra stringhe per il pattern `ababbabbababbababbabb` sull'alfabeto $\Sigma = \{a, b\}$.

34.3-3 Un pattern P è detto ***non sovrapponibile*** se $P_i \sqsupset P_q$ implica $k = 0$ o $k = q$. Si descriva il diagramma di transizione di stati per un automa di corrispondenza tra stringhe per un pattern non sovrapponibile.

** 34.3-4* Si descriva come costruire un automa a stati finiti che, dati due pattern P e P', determini tutte le occorrenze di *entrambi* i pattern. Si cerchi di minimizzare il numero di stati dell'automa.

34.3-5 Dato un pattern P contenente caratteri jolly (si veda l'Esercizio 34.1-5), si mostri come costruire un automa a stati finiti in grado di tovare un'occorrenza di P in un testo T in tempo $O(n)$, dove $n = |T|$.

34.4 Algoritmo Knuth-Morris-Pratt

Si presenta ora un algoritmo per la corrispondenza tra stringhe dovuto a Knuth, Morris e Pratt. Questo algoritmo raggiunge tempo di esecuzione $\Theta(n + m)$ evitando del tutto il calcolo della funzione di transizione δ ed esegue il controllo del pattern usando solo una funzione ausiliaria $\pi[1..m]$ precalcolata in base al pattern in tempo $O(m)$. L'array π permette che la funzione di transizione δ venga calcolata efficientemente "al volo" quando necessario (il calcolo è ammortizzato). In parole povere, per qualunque stato $q = 0, 1, \ldots, m$ e qualsiasi carattere $a \in \Sigma$, il valore $\pi[q]$ contiene informazioni indipendenti da a che sono necessarie per calcolare $\delta(q, a)$. (Quest'aspetto sarà più chiaro tra breve.) Poiché l'array π ha solo m elementi, mentre δ ha $O(m|\Sigma|)$ elementi, nella preelaborazione si risparmia un fattore $|\Sigma|$ calcolando π invece di δ.

Funzione prefisso per un pattern

La funzione prefisso per un pattern racchiude la conoscenza di come il pattern compare all'interno di se stesso. Quest'informazione può essere usata per evitare di controllare inutili spostamenti nell'algoritmo ingenuo di corrispondenza tra stringhe o per evitare la preelaborazione di δ per un automa a stati finiti.

Si consideri il comportamento dell'algoritmo ingenuo. La figura 34.9(a) mostra un particolare spostamento s di una configurazione contenente il pattern P = `ababaca` nel testo T. Per quest'esempio, $q = 5$ caratteri combaciano, ma il sesto carattere del pattern non combacia col carattere nel testo. La conoscenza di questi q caratteri del testo consente di determinare immediatamente che certi spostamenti non sono validi. Nell'esempio della figura, lo spostamento $s + 1$ è necessariamente non valido, poiché il primo carattere del pattern, una `a`, dovrebbe essere allineato con un carattere del testo che si sa corrisponde al secondo carattere del pattern, cioè una `b`. Tuttavia, lo spostamento $s + 2$ mostrato nella parte *b* della figura allinea i primi tre caratteri del pattern con tre caratteri del testo che devono necessariamente combaciare. In generale, è utile conoscere la risposta alla seguente domanda:

Dato che i caratteri del pattern $P[1..q]$ combaciano con i caratteri del testo $T[s + 1..s + q]$, qual è il minimo spostamento $s' > s$ tale che

$$P[1..k] = T[s' + 1..s' + k], \tag{34.5}$$

dove $s' + k = s + q$?

Un tale spostamento s' è il primo spostamento maggiore di s che, in base alla conoscenza su $T[s + 1..s + q]$, non sia sicuramente non valido. Nel migliore dei casi, si ha che $s' = s + q$ e gli spostamenti $s + 1, s + 2, \ldots, s + q - 1$ sono esclusi tutti immediatamente. In ogni caso,

Figura 34.9 *La funzione prefisso π.* (a) *Il pattern P =* ababaca *è allineato con un testo T in modo che i primi $q = 5$ caratteri combacino. I caratteri combacianti, evidenziati in grigio, sono connessi con linee verticali.* (b) *Usando solo la conoscenza dei 5 caratteri combacianti, si può dedurre che uno spostamento $s + 1$ non è valido, ma che uno spostamento $s' = s + 2$ è consistente con ciò che si conosce del testo ed è quindi potenzialmente valido.* (c) *Le informazioni utili per tali deduzioni possono essere precalcolate confrontando il pattern con se stesso. In questo caso si nota che il prefisso più lungo di P che sia anche un suffisso di P_5 è P_3. Quest'informazione è precalcolata e rappresentata nell'array π, così che $\pi[5] = 3$. Dato che q caratteri corrispondono con successo allo spostamento s, il prossimo spostamento potenzialmente valido è $s' = s + (q - \pi[q])$.*

nel nuovo spostamento s' non sarà necessario confrontare i primi k caratteri di P con i corrispondenti caratteri di T poiché si ha la garanzia, per l'equazione (34.5), che essi combaciano.

L'informazione necessaria può essere precalcolata confrontando il pattern con sé stesso, come mostrato nella figura 34.9(c). Poiché $T[s' + 1 .. s' + k]$ è parte della porzione conosciuta del testo, esso è un suffisso della stringa P_q. L'equazione (34.5) può quindi essere interpretata come una richiesta del più grande $k < q$ tale che $P_k \sqsupset P_q$. Allora, $s' = s + (q - k)$ è il prossimo spostamento potenzialmente valido. Sembra che sia conveniente memorizzare il numero k di caratteri corrispondenti al nuovo spostamento s', piuttosto che memorizzare $s' - s$. Quest'informazione può essere usata per rendere più veloce sia l'algoritmo ingenuo di corrispondenza tra stringhe, sia l'automa a stati finiti di corrispondenza tra stringhe.

Si formalizza il precalcolo richiesto come segue. Dato un pattern $P[1..m]$, la ***funzione prefisso*** per il pattern P è la funzione $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m - 1\}$ tale che

$$\pi[q] = \max\{k : k < q \text{ e } P_k \sqsupset P_q\} .$$

Cioè, $\pi[q]$ è la lunghezza del prefisso più lungo di P che sia un suffisso proprio di P_q. Come altro esempio, la figura 34.10(a) fornisce la funzione prefisso π completa per il pattern ababababca.

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

(a)

Figura 34.10 *Un'illustrazione del lemma 34.5 per il pattern P =* ababababca *e* $q = 8$. (a) *La funzione π per il pattern dato. Poiché $\pi[8] = 6$, $\pi[6] = 4$, $\pi[4] = 2$ e $\pi[2] = 0$, iterando π si ottiene $\pi^*[8] = \{8, 6, 4, 2, 0\}$.* (b) *Si fa scorrere la configurazione contenente il pattern P verso destra e si annota quando qualche prefisso P_k di P combacia con qualche prefisso proprio di P_8; ciò si verifica per $k = 6, 4, 2$ e 0. Nella figura, la prima riga mostra P e la linea punteggiata verticale è tirata proprio dopo P_8. Le righe successive mostrano tutti gli spostamenti di P per cui qualche prefisso P_k di P combacia con qualche suffisso di P_8. I caratteri che combaciano sono mostrati in grigio. Le linee verticali connettono i caratteri allineati che combaciano. Pertanto, $\{k : P_k \sqsupset P_q\} = \{8, 6, 4, 2, 0\}$. Il lemma afferma che $\pi^*[q] = \{k : P_k \sqsupset P_q\}$ per tutte le q.*

L'algoritmo di corrispondenza di Knuth-Morris-Pratt è dato nel seguente pseudocodice con nome KMP-MATCHER. È principalmente basato, come si vedrà, su FINITE-AUTOMATON-MATCHER. KMP-MATCHER chiama la procedura ausiliaria COMPUTE-PREFIX-FUNCTION per calcolare π.

```
KMP-MATCHER (T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0
for i ← 1 to n
    do while q > 0 e P[q + 1] ≠ T[i]
            do q ← π[q]
        if P[q + 1] = T[i]
            then  q ← q + 1
        if q = m
            then  stampa "Il pattern appare con spostamento" i − m
                  q ← π[q]
```

```
COMPUTE-PREFIX-FUNCTION(P)
1  m ← length[P]
2  π[1] ← 0
3  k ← 0
4  for q ← 2 to m
5      do while k > 0 e P[k + 1] ≠ P[q]
6             do k ← π[k]
7         if P[k + 1] = P[q]
8             then  k ← k + 1
9         π[q] ← k
10 return π
```

Si comincia con un'analisi del tempo di esecuzione di queste procedure. Dimostrarne la correttezza sarà più complicato.

Analisi del tempo di esecuzione

Mediante analisi ammortizzata (si veda Capitolo 18) si mostra che il tempo di esecuzione di COMPUTE-PREFIX-FUNCTION è $O(m)$. Si associa il potenziale di k allo stato corrente k dell'algoritmo. Questo potenziale ha un valore iniziale 0 alla linea 3. Quando è eseguita la linea 6, k viene decrementato poichè $\pi[k] < k$. Poiché però, $\pi[k] \geq 0$ per ogni k, k non può diventare mai negativo. L'unica altra linea che modifica k è la linea 8 che incrementa k di 1 durante ogni esecuzione del corpo del ciclo **for**. Poiché $k < q$ quando comincia il ciclo **for** e poiché q è incrementato di 1 ad ogni iterazione del corpo del ciclo **for**, vale sempre $k < q$. (Ciò giustifica l'affermazione che anche $\pi[q] < q$ alla linea 9.) Si può pagare ogni esecuzione del corpo del ciclo **while** alla linea 6 decrementando corrispondentemente la funzione potenziale, poiché $\pi[k]<k$. Alla linea 8 si incrementa la funzione potenziale al più di 1, così che il costo ammortizzato del corpo del ciclo alle linee 5–9 è $O(1)$. Poiché il numero di iterazioni del ciclo più esterno è $O(m)$ e poichè la funzione potenziale finale è grande almeno quanto la funzione potenziale iniziale, il tempo di esecuzione totale effettivo di COMPUTE-PREFIX-FUNCTION è $O(m)$ nel caso peggiore.

L'algoritmo di Knuth-Morris-Pratt impiega tempo $O(n + m)$. La chiamata di COMPUTE-PREFIX-FUNCTION richiede tempo $O(m)$, come si è appena visto, e un'analisi ammortizzata analoga, usando il valore di q come funzione potenziale, mostra che la restante parte di KMP-MATCHER richiede tempo $O(n)$.

In confronto a FINITE-AUTOMATON-MATCHER, usando π invece di δ, si può ridurre il tempo di preelaborazione del pattern da $O(m|\Sigma|)$ a $O(m)$, continuando a mantenere il tempo per individuare le occorrenze limitato da $O(m + n)$.

Correttezza del calcolo della funzione prefisso

Si comincia con un lemma essenziale che mostra che iterando la funzione prefisso π, si possono elencare tutti i prefissi P_k che siano suffissi di un dato prefisso P_q. Sia

$$\pi^*[q] = \{q, \pi[q], \pi^2[q], \pi^3[q], \ldots, \pi^u[q]\},$$

dove $\pi^i[q]$ è definito in termini della composizione funzionale, cioè $\pi^0[q] = q$ e $\pi^{i-1}[q] = \pi[\pi^i[q]]$ per $i > 1$, e dove è chiaro che la sequenza in $\pi^*[q]$ termina quando è raggiunto $\pi^i[q] = 0$.

Lemma 34.5 (Lemma dell'iterazione della funzione prefisso)

Sia P un pattern di lunghezza m con funzione prefisso π. Allora per $q = 1, 2, \ldots, m$, si ha: $\pi^*[q] = \{k : P_k \sqsupset P_q\}$.

Dimostrazione. Si dimostra prima che

$$i \in \pi^*[q] \text{ implica } P_i \sqsupset P_q. \tag{34.6}$$

Se $i \in \pi^*[q]$, allora $i = \pi^u[q]$ per qualche u. Si dimostra la validità dell'equazione (34.6) per induzione su u. Per $u = 0$, si ha $i = q$, e l'affermazione è vera poiché $P_q \sqsupset P_q$. Usando la relazione $P_{\pi[i]} \sqsupset P_i$ e la transitività di $\sqsupset$, si dimostra l'affermazione per tutti gli i in $\pi^*[q]$. Per cui $\pi^*[q] \subseteq \{k : P_k \sqsupset P_q\}$.

Si mostra ora per assurdo che $\{k : P_k \sqsupset P_q\} \subseteq \pi^*[q]$. Si supponga per assurdo che vi sia almeno un intero nell'insieme $\{k : P_k \sqsupset P_q\} - \pi^*[q]$ e sia j il più grande di tali valori. Poiché q è in $\{k : P_k \sqsupset P_q\} \cap \pi^*[q]$, allora $j < q$; si denoti con j' il più piccolo intero in $\pi^*[q]$ maggiore di j. (Si può scegliere $j' = q$ se non vi è un altro numero in $\pi^*[q]$ che sia più grande di j.) Si ha $P_j \sqsupset P_q$ perché $j \in \{k : P_k \sqsupset P_q\}$, $P_{j'} \sqsupset P_q$ perché $j' \in \pi^*[q]$; perciò $P_j \sqsupset P_{j'}$ per il Lemma 34.1. Inoltre, j è il più grande valore con questa proprietà. Quindi si deve avere $\pi[j'] = j$ e così $j \in \pi^*[q]$. Questa contraddizione dimostra il lemma. ■

La figura 34.10 illustra questo lemma.

L'algoritmo COMPUTE-PREFIX-FUNCTION calcola $\pi[q]$, per $q = 1, 2, \ldots, m$, nell'ordine. Il calcolo di $\pi[1] = 0$ nella linea 2 di COMPUTE-PREFIX-FUNCTION è certamente corretto, poiché $\pi[q] < q$, per tutte le q. Il seguente lemma e il suo corollario saranno usati per dimostrare che COMPUTE-PREFIX-FUNCTION calcola correttamente $\pi[q]$ per $q > 1$.

Lemma 34.6

Sia P un pattern di lunghezza m e sia π la funzione prefisso per P. Per $q = 1, 2, \ldots, m$, se $\pi[q] > 0$, allora $\pi[q] - 1 \in \pi^*[q - 1]$.

Dimostrazione. Se $k = \pi[q] > 0$, allora $P_k \sqsupset P_q$ e quindi $P_{k-1} \sqsupset P_{q-1}$ (ignorando l'ultimo carattere di P_k e P_q). Dal lemma 34.5, allora, $k - 1 \in \pi^*[q - 1]$. ■

Per $q = 2, 3, \ldots, m$, si definisce il sottoinsieme $E_{q-1} \subseteq \pi^*[q - 1]$ come

$$E_{q-1} = \{k : k \in \pi^*[q - 1] \text{ e } P[k + 1] = P[q]\}.$$

L'insieme E_{q-1} consiste dei valori di k per cui $P_k \sqsupset P_{q-1}$ (lemma 34.5); poiché $P[k + 1] = P[q]$, è vero anche che, per questi valori di k, $P_{k+1} \sqsupset P_q$. Intuitivamente E_{q-1} consiste di quei valori $k \in \pi^*[q - 1]$ tali che si può estendere P_k a P_{k+1} e ottenere un suffisso di P_q.

Corollario 34.7

Sia P un pattern di lunghezza m e sia π la funzione prefisso per P. Per $q = 2, 3, \ldots, m$,

$$\pi[q] = \begin{cases} 0 & \text{se } E_{q-1} = 0 , \\ 1 + \max\{k \in E_{q-1}\} & \text{se } E_{q-1} \neq 0 . \end{cases}$$

Dimostrazione. Se $r = \pi[q]$, allora $P_r \sqsupset P_q$, quindi $r \geq 1$ implica $P[r] = P[q]$. Dal Lemma 34.6 segue che, se $r \geq 1$, allora

$$r = 1 + \max\{k \in \pi^*[q-1] : P[k+1] = P[q]\} .$$

Ma l'insieme massimizzato è proprio E_{q-1}, così che $r = 1 + \max\{k \in E_{q-1}\}$ e E_{q-1} è non vuoto. Se $r = 0$, non vi è alcun $k \in \pi^*[q-1]$ per cui si può estendere P_k a P_{k+1} e ottenere un suffisso di P_q, poiché allora si dovrebbe avere $\pi[q] > 0$. Pertanto, $E_{q-1} = 0$. ■

Si può ora completare la dimostrazione che COMPUTE-PREFIX-FUNCTION calcola π correttamente. Nella procedura COMPUTE-PREFIX-FUNCTION, all'inizio di ogni iterazione del ciclo for alle linee 4-9, si ha che $k = \pi[q-1]$. Questa condizione è verificata alle linee 2 e 3 quando il ciclo inizia per la prima volta e rimane vera ad ogni successiva iterazione in virtù della linea 9. Le linee 5-8 fanno sì che k diventi il valore corretto di $\pi[q]$. Il ciclo alle linee 5-6 cerca tra tutti i valori di $k \in \pi^*[q-1]$ finché ne trova uno per cui $P[k+1] = P[q]$; a quel punto si ha che k è il più grande valore nell'insieme E_{q-1}, così che, dal corollario 34.7, si può assegnare il valore $k + 1$ a $\pi[q]$. Se un tale valore di k non viene trovato, $k = 0$ alle linee 7-9 e a $\pi[q]$ è assegnato 0. Ciò completa la dimostrazione della correttezza di COMPUTE-PREFIX-FUNCTION.

Correttezza dell'algoritmo KMP

La procedura KMP-MATCHER può essere vista come una realizzazione rivisitata della procedura FINITE-AUTOMATON-MATCHER. In particolare, si mostrerà che il codice alle linee 6-9 di KMP-MATCHER è equivalente alla linea 4 di FINITE-AUTOMATON-MATCHER che assegna $\delta(q, T[i])$ a q. Invece di usare un valore memorizzato di $\delta(q, T[i])$, però, questo valore è ricalcolato quando necessario a partire da π. Una volta compreso che KMP-MATCHER simula il comportamento di FINITE-AUTOMATON-MATCHER, la correttezza di KMP-MATCHER segue dalla correttezza di FINITE-AUTOMATON-MATCHER (sebbene si vedrà fra poco perché sia necessaria la linea 12 in KMP-MATCHER).

La correttezza di KMP-MATCHER segue dall'affermazione che $\delta(q, T[i]) = 0$ oppure $\delta(q, T[i]) - 1 \in \pi^*[q]$. Per verificare quest'affermazione, sia $k = \delta(q, T[i])$. Allora $P_k \sqsupset P_q T[i]$ per le definizioni di δ e σ. Quindi $k = 0$ oppure $k \geq 1$ e $P_{k-1} \sqsupset P_q$, trascurando l'ultimo carattere sia di P_k che di $P_q T[i]$ (nel qual caso $k - 1 \in \pi^*[q]$). Quindi, $k = 0$ oppure $k - 1 \in \pi^*[q]$, dimostrando l'affermazione.

L'affermazione è usata come segue. Si denoti con q' il valore di q quando si esegue la linea 6 per la prima volta. Si usa l'equivalenza $\pi^*[q] = \{k : P_k \sqsupset P_q\}$ per giustificare l'iterazione $q \leftarrow \pi[q]$ che enumera gli elementi di $\{k : P_k \sqsupset P_{q'}\}$. Le linee 6-9 determinano $\delta(q', T[i])$ esaminando gli elementi di $\pi^*[q']$ in ordine decrescente. Il codice usa l'affermazione per cominciare con $q = \phi(T_{i-1}) = \sigma(T_{i-1})$ ed eseguire l'iterazione $q \leftarrow \pi[q]$ finché viene trovato un q tale che $q = 0$ o $P[q+1] = T[i]$. Nel primo caso, $\delta(q', T[i]) = 0$; nell'ultimo caso q è l'elemento massimo di $E_{q'}$, così che $\delta(q', T[i]) = q + 1$ per il corollario 34.7.

La linea 12 è necessaria in KMP-MATCHER per evitare un possibile riferimento a $P[m+1]$ alla linea 6 dopo che è stata trovata un'occorrenza di P. (L'argomentazione che $q = \sigma(T_{i-1})$ prima della successiva esecuzione della linea 6 rimane valida per il suggerimento dell'Esercizio 34.4-6: $\delta(m, a) = \delta(\pi[m], a)$ oppure, equivalentemente, $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ per qualunque $a \in \Sigma$.) Le altre argomentazioni sulla correttezza dell'algoritmo di Knuth-Morris-Pratt seguono dalla correttezza di FINITE-AUTOMATON-MATCHER, poiché, come si è appena visto, KMP-MATCHER simula il comportamento di FINITE-AUTOMATON-MATCHER.

Esercizi

34.4-1 Si calcoli la funzione prefisso π per il pattern ababbabbababbababbabb quando l'alfabeto è $\Sigma = \{a, b\}$.

34.4-2 Si dia un limite superiore alla dimensione di $\pi^*[q]$ in funzione di q. Si dia un esempio per mostrare che il limite è stretto.

34.4-3 Si spieghi come determinare le occorrenze del pattern P nel testo T esaminando la funzione π per la stringa PT (la stringa di lunghezza $m + n$ che è la concatenazione di P e T).

34.4-4 Si mostri come migliorare KMP-MATCHER sostituendo le occorrenze di π alla linea 7 (ma non alla linea 12) con π', dove π' è definito ricorsivamente, per $q = 1, 2, \ldots, m$, dall'equazione

$$\pi'[q] = \begin{cases} 0 & \text{se } \pi[q] = 0 , \\ \pi'[\pi[q]] & \text{se } \pi[q] \neq 0 \text{ e } P[\pi[q]+1] = P[q+1] , \\ \pi[q] & \text{se } \pi[q] \neq 0 \text{ e } P[\pi[q]+1] \neq P[q+1] . \end{cases}$$

Si spieghi perché l'algoritmo modificato è corretto e si spieghi in che senso queste modifiche costituiscono un miglioramento.

34.4-5 Si dia un algoritmo lineare in tempo per determinare se un testo T è una rotazione ciclica di un'altra stringa T'. Per esempio, `arc` e `car` sono rotazioni cicliche l'una dell'altra.

* *34.4-6* Si dia un algoritmo efficiente per calcolare la funzione di transizione δ per l'automa di corrispondenza tra stringhe associato a un dato pattern P. L'algoritmo dovrebbe richiedere tempo $O(m|\Sigma|)$. (*Suggerimento*: si dimostri che $\delta(q, a) = \delta(\pi[q], a)$ se $q = m$, oppure $P[q+1] \neq a$.)

* 34.5 Algoritmo Boyer-Moore

Se il pattern P è relativamente lungo e l'alfabeto Σ è ragionevolmente grande, allora l'algoritmo proposto da Robert S. Boyer e J. Strother Moore risulta essere probabilmente il più efficiente algoritmo per la corrispondenza fra stringhe.

```
BOYER-MOORE-MATCHER(T,P,Σ)
1  n ← length[T]
2  m ← length[P]
3  λ ← COMPUTE-LAST-OCCURRENCE-FUNCTION(P,m,Σ)
4  γ ← COMPUTE-GOOD-SUFFIX-FUNCTION(P,m)
5  s ← 0
6  while s ≤ n−m
7      do j ← m
8         while j > 0 e P[j] = T[s + j]
9             do j ← j−1
10        if j = 0
11            then  stampa "Il pattern appare con spostamento" s
12                s ← s + γ[0]
13            else  s ← s + max(γ[j], j − λ[T[s + j]])
```

A parte l'aspetto misterioso di λ e γ, questo programma somiglia molto all'algoritmo ingenuo di corrispondenza tra stringhe. Infatti, si supponga di commentare le linee 3-4 e di sostituire l'aggiornamento di s alle linee 12-13 con semplici incrementi come segue:

```
12                s ← s + 1
13            else  s ← s + 1
```

Il programma modificato si comporta adesso esattamente come il confrontatore di stringhe ingenuo: il ciclo **while** che comincia alla linea 6 considera uno dopo l'altro ciascuno degli $n - m + 1$ possibili spostamenti s ed il ciclo **while** che comincia alla linea 8 verifica la condizione $P[1 .. m] = T[s + 1 .. s + m]$ confrontando $P[j]$ con $T[s + j]$, dove $j = m, m - 1, ..., 1$.

Se il ciclo termina con $j = 0$, è stato trovato uno spostamento s valido e la linea 11 restituisce il valore di s. A questo livello, le uniche caratteristiche rimarchevoli dell'algoritmo di Boyer e Moore sono che esso confronta il pattern con il testo *da destra verso sinistra* e che incrementa lo spostamento s alle linee 12-13 di un valore che non è necessariamente 1.

L'algoritmo di Boyer e Moore usa due euristiche che gli consentono di evitare la maggior parte del lavoro che invece eseguiva il precedente algoritmo per confrontare le stringhe. Queste euristiche sono così determinanti che spesso permettono all'algoritmo di saltare in un colpo solo l'esame di molti caratteri del testo. Queste euristiche, conosciute con i nomi di "euristica del carattere discordante" ed "euristica del buon suffisso", sono illustrate nella figura 34.11. Possono essere viste come operanti in parallelo e indipendentemente. Quando si verifica una discordanza, ogni euristica propone una quantità di cui s possa essere incrementata con sicurezza senza saltare una posizione valida. L'algoritmo Boyer-Moore sceglie la quantità più grande e incrementa s di quella quantità: quando è raggiunta la linea 13 dopo una discordanza, l'euristica del carattere discordante propone di incrementare s di $j - \lambda[T[s + j]]$, mentre l'euristica del buon suffisso propone un incremento di $\gamma[j]$.

Figura 34.11 *Un'illustrazione dell'euristica Boyer-Moore.* (a) *Corrispondenza tra il pattern* `reminescence` *e un testo attraverso i confronto dei caratteri da destra a sinistra. Lo spostamento s non è valido, sebbene un "buon suffisso"* `ce` *del pattern combaci con i rispettivi caratteri del testo (i caratteri combaciamti sono mostrati in grigio), viene scoperto il "carattere discordante"* `i` *che non combacia con il corrispondente carattere* `n` *del pattern.* (b) *L'euristica del carattere discordante propone di spostare il pattern a destra, se possibile, della quantità che garantisca che il carattere discordante del testo combaci con l'occorrenza più a destra del carattere discordante del pattern. In questo esempio, se si sposta il pattern di 4 posizioni a destra allora il carattere discordante* `i` *del testo combacia con il carattere* `i` *più a destra del pattern nella posizione 6. Se il carattere discordante non appare nel pattern, allora il pattern può essere spostato del tutto dopo il carattere discordante del testo. Se l'occorrenza più a destra del carattere discordante nel pattern è a destra della posizione corrente del carattere discordante, allora questa euristica non propone niente.* (c) *Con l'euristica del buon suffisso, il pattern è spostato a destra della quantità minima che garantisca che i caratteri del pattern che si allineano con il buon suffisso* `ce`, *trovato precedentemente nel testo, combaceranno con questi caratteri del suffisso. In questo esempio, se si sposta il pattern di 3 posizioni a destra questa condizione viene soddisfatta. Poiché l'euristica del buon suffisso propone di spostare il pattern di 3 posizioni, che è un numero più piccolo di quello proposto dall'euristica del carattere discordante di 4 posizioni, l'algoritmo Boyer-Moore incrementa lo spostamento di 4.*

L'euristica del carattere discordante

Quando si verifica una discordanza, l'euristica del carattere discordante usa informazioni su dove il carattere discordante del testo $T[s + j]$ appare nel pattern (o se non appare affatto) per proporre un nuovo spostamento. Nel caso migliore, la discordanza si verifica sul primo confronto ($P[m] \neq T[s + m]$) e il carattere discordante $T[s + m]$ non appare affatto nel pattern. (Si immagini di cercare "a" nella stringa di testo "b".) In questo caso si può incrementare lo spostamento s di m, poiché qualunque spostamento più piccolo di $s + m$ allineerà qualche carattere del pattern con il carattere discordante. Se il caso migliore si verifica ripetutamente,

l'algoritmo Boyer-Moore esamina solo una frazione $1/m$ di caratteri del testo, poiché ogni carattere esaminato del testo porta a una discordanza, così da incrementare s di m. Il comportamento nel caso migliore illustra la potenza del confronto da destra a sinistra rispetto a quello da sinistra a destra.

In generale, l'***euristica del carattere discordante*** funziona come segue. Si supponga di avere appena trovato una discordanza: $P[j] \neq T[s+j]$ per qualche j, dove $1 \leq j \leq m$. Sia allora k l'indice più grande nell'intervallo $1 \leq k \leq m$ tale che $T[s+j] = P[k]$, se un tale k esiste, altrimenti sia $k = 0$. Si può incrementare, con successo, s di $j - k$. Per dimostrare questa affermazione si devono considerare tre casi, come illustrato nella figura 34.12.

- $k = 0$: come mostrato nella figura 34.12(a), il carattere discordante $T[s+j]$ non appare affatto nel pattern, così si può incrementare con successo s di j senza saltare alcuno spostamento valido.
- $k < j$: come mostrato nella figura 34.12(b), l'occorrenza più a destra del carattere discordante è, nel pattern, a sinistra della posizione j, così che $j - k > 0$ e il pattern deve essere spostato di $j - k$ caratteri a destra prima che il carattere discordante del testo combaci con qualche carattere del pattern. Quindi, si può, con successo, incrementare s di $j - k$ senza saltare alcuno spostamento valido.
- $k > j$: come mostrato nella figura 34.12(c), $j - k < 0$ e così l'euristica del carattere discordante propone essenzialmente di decrementare s. Questa proposta sarà ignorata dall'algoritmo Boyer-Moore, poiché l'euristica del buon suffisso proporrà uno spostamento a destra in tutti i casi.

Il semplice programma che segue definisce $\lambda[a]$ come l'indice della posizione, nel pattern, più a destra in cui appare il carattere a, per ogni carattere $a \in \Sigma$. Se a non appare nel pattern, allora a $\lambda[a]$ è assegnato 0. La funzione λ è detta ***funzione dell'ultima occorrenza*** nel pattern. Con questa definizione, l'espressione $j - \lambda[T[s+j]]$ alla linea 13 di BOYER-MOORE MATCHER realizza l'euristica del carattere discordante. (Poiché $j - \lambda[T[s+j]]$ è negativo se l'occorrenza più a destra del carattere discordante $T[s+j]$ è, nel pattern, a destra della posizione j, per assicurare che l'algoritmo faccia progressi ad ogni passo, in tali casi ci si basa sulla positività di $\gamma[j]$ derivante dall'euristica del buon suffisso.)

```
COMPUTE-LAST-OCCURENCE-FUNCTION
1  for ogni carattere a ∈ Σ
2      do λ[a] ← 0
3  for j ← 1 to m
4      do λ[P[j]] ← j
5  return λ
```

Il tempo di esecuzione della procedura COMPUTE-LAST-OCCURENCE-FUNCTION è $O(|\Sigma| + m)$.

Euristica del buon suffisso

Sia definita la relazione $Q \sim R$ (si legga "Q è simile a R") sulle stringhe Q e R per indicare che $Q \sqsupset R$ o $R \sqsupset Q$. Se due stringhe sono simili, allora esse possono essere allineate avendo i loro caratteri più a destra combaciantì e nessuna coppia di caratteri allineati sarà discorde. La relazione "$\sim$" è simmetrica: $Q \sim R$ se e solo se $R \sim Q$. Come conseguenza del lemma 34.1, si ha anche che

Figura 34.12 *I casi dell'euristica del carattere discordante.* (a) *Il carattere discordante* h *non appare nel pattern, così il pattern può avanzare di* $j = 11$ *caratteri oltrepassando il carattere discordante.* (b) *L'occorrenza più a destra del carattere discordante, nel pattern, è nella posizione* $k < j$, *così il pattern può avanzare di* $j - k$ *caratteri. Poiché* $j = 10$ *e* $k = 6$ *per il carattere discordante* i, *il pattern può avanzare di 4 posizioni finché la* i *sia allineata.* (c) *L'occorrenza più a destra discordante del carattere, nel pattern, è alla posizione* $k > j$. *In questo esempio,* $j = 10$ *e* $k = 12$ *per il carattere discordante* e. *L'euristica del carattere discordante propone uno spostamento negativo che viene ignorato.*

$$Q \sqsupset R \text{ e } S \sqsupset R \text{ implicano } Q \sim S. \tag{34.7}$$

Se si trova che $P[j] \neq T[s+j]$, dove $j < m$, allora ***l'euristica del buon suffisso*** dice che si può, con successo, spostare s di

$$\gamma[j] = m - \max\{k : 0 \leq k < m \text{ e } P[j+1 \,..\, m] \sim P_k\}.$$

Cioè, $\gamma[j]$ è la quantità minima di cui s può avanzare senza far sì che alcun carattere nel "buon suffisso" $T[s + j + 1 .. s + m]$ sia discorde dal nuovo allineamento del pattern. La funzione γ è ben definita per tutte le j, poiché $P[j + 1 .. m] \sim P_0$, per tutte le j: la stringa vuota è simile a tutte le stringhe. La funzione γ è detta ***funzione del buon suffisso*** per il pattern P.

Si mostra ora come calcolare la funzione γ del buon suffisso. Anzitutto si dimostra che $\gamma[j] \le m - \pi[m]$ per tutte le j, come segue. Se $w = \pi[m]$, allora $P_w \sqsupset P$ per definizione di π. Inoltre, poiché $P[j+1 .. m] \sqsupset P$ per qualunque j, si ha $P_w \sim P[j+1 .. m]$ dall'equazione (34.7). Quindi, $\gamma[j] \le m - \pi[m]$ per tutte le j.

Si può ora riscrivere la definizione di γ come

$$\gamma[j] = m - \max\{k : \pi[m] \le k < m \text{ e } P[j + 1 .. m] \sim P_k\} .$$

La condizione $P[j + 1 .. m] \sim P_k$ vale se $P[j + 1 .. m] \sqsupset P_k$ o anche se $P_k \sqsupset P[j + 1 .. m]$. Ma l'ultima possibilità implica che $P_k \sqsupset P$ e pertanto che $k \le \pi[m]$, dalla definizione di π. Quest'ultima possibilità non può ridurre il valore di $\gamma[j]$ al di sotto di $m - \pi[m]$. Si può dunque riscrivere ulteriormente la definizione di γ come segue:

$$\gamma[j] = m - \max(\{\pi[m]\} \cup \{k : \pi[m] < k < m \text{ e } P[j + 1 .. m] \sqsupset P_k\}) .$$

(Il secondo insieme può essere vuoto.) Si osservi che la definizione implica $\gamma[j] > 0$ per tutte le $j = 1, 2, \ldots, m$, il che assicura che l'algoritmo Boyer-Moore faccia progressi.

Per semplificare ulteriormente l'espressione per γ, si definisce P' come l'inverso del pattern P e π' come la funzione prefisso corispondente. Cioè, $P'[i] = P[m - i + 1]$ per $i = 1, 2, \ldots, m$ e $\pi'[t]$ è il più grande u tale che $u < t$ e $P'_u \sqsupset P'_t$.

Se k è il valore più grande possibile tale che $P[j + 1..m] \sqsupset P_k$, allora si afferma che

$$\pi'[l] = m - j , \tag{34.8}$$

dove $l = (m - k) + (m - j)$. Per vedere che quest'affermazione è ben definita, si noti che $P[j + 1 .. m] \sqsupset P_k$ implica che $m - j < k$ e, pertanto, $l \le m$. Inoltre $j < m$ e $k \le m$, quindi $l \ge 1$. Si dimostra quest'affermazione come segue. Poiché $P[j + 1 .. m] \sqsupset P_k$, si ha $P'_{m-j} \sqsupset P'_l$. Quindi, $\pi'[l] \ge m - j$. Si supponga ora che $p > m - j$, dove $p = \pi'[l]$. Quindi, dalla definizione di π', si ha $P'_p \sqsupset P'_l$ o, equivalentemente, $P'[1 .. p] = P'[l - p + 1 .. l]$. Riscrivendo quest'equazione in termini di P piuttosto che di P', si ha $P[m - p + 1..m] = P[m - l + 1 .. m - l + p]$. Sostituendo $l = 2m - k - j$, si ottiene $P[m - p + 1 .. m] = P[k - m + j + 1 .. k - m + j + p]$ che implica $P[m - p + 1 .. m] \sqsupset P_{k-m+j+p}$. Poiché $p > m - j$, si ha $j + 1 > m - p + 1$ e così $P[j + 1 .. m] \sqsupset P[m - p + 1 .. m]$ che implica $P[j + 1 .. m] \sqsupset P_{k-m+j+p}$, per la transitività di $\sqsupset$. Infine, poiché $p > m - j$, si ha $k' > k$, dove $k' = k - m + j + p$, contraddicendo la scelta di k come il più grande valore possibile tale che $P[j + 1 .. m] \sqsupset P_k$. Questa contraddizione significa che non si può avere $p > m - j$, pertanto $p = m - j$, il che dimostra l'affermazione (34.8).

Usando l'equazione (34.8), e notando che $\pi'[l] = m - j$ implica $j = m - \pi'[l]$ e $k = m - l + \pi'[l]$, si può ancora una volta riscrivere la definizione di γ:

$$\begin{aligned}\gamma[j] &= m - \max(\{\pi[m]\} \cup \{m - l + \pi'[l] : 1 \le l \le m \text{ e } j = m + \pi'[l]\}) \\ &= \min(\{m - \pi[m]\} \cup \{l + \pi'[l] : 1 \le l \le m \text{ e } j = m + \pi'[l]\})\end{aligned} \tag{34.9}$$

Anche in questo caso il secondo insieme può essere vuoto.

A questo punto si può esaminare la procedura per calcolare γ.


```
COMPUTE-GOOD-SUFFIX-FUNCTION(P,m)
π ← COMPUTE-PREFIX-FUNCTION(P)
P' ← reverse(P)
π' ← COMPUTE-PREFIX-FUNCTION(P')
for j ← 0 to m
    do γ[j] ← m − π[m]
for l ← 1 to m
    do j ← m − π'[l]
       if γ[j] > l − π'[l]
          then γ[j] ← l − π'[l]
return γ
```

La procedura COMPUTE-GOOD-SUFFIX-FUNCTION è una semplice realizzazione dell'equazione (34.9). Il suo tempo di esecuzione è $O(m)$.

Il tempo di esecuzione nel caso peggiore dell'algoritmo Boyer-Moore è chiaramente $O((n - m + 1)m + |\Sigma|)$, poiché COMPUTE-LAST-OCCURRENCE-FUNCTION richiede tempo $O(m + |\Sigma|)$, COMPUTE-GOOD-SUFFIX-FUNCTION richiede tempo $O(m)$ e l'algoritmo Boyer-Moore (come l'algoritmo Rabin-Karp) impiega tempo $O(m)$ convalidando ogni spostamento s valido. Tuttavia, in pratica l'algoritmo Boyer-Moore è spesso il più usato.

Esercizi

34.5-1 Si calcolino le funzioni λ e γ per il pattern P = `0101101201` e l'alfabeto $\Sigma = \{0, 1, 2\}$.

34.5-2 Si fornisca un esempio per mostrare che combinando le euristiche del carattere discordante e del buon suffisso, l'algoritmo Boyer-Moore può comportarsi molto meglio che usando solo l'euristica del buon suffisso.

* *34.5-3* Un miglioramento della procedura Boyer-Moore che è spesso usata in pratica è di sostituire la funzione γ con γ', definita da

$$\begin{aligned}\gamma'[j] = m - \max\{k : 0 \le k < m \text{ e } P[j + 1 .. m] \sim P_k \text{ e} \\ (k - m + j > 0 \text{ implica } P[j] \ne P[k - m + j])\}.\end{aligned}$$

Oltre che assicurare che i caratteri nel buon suffisso saranno discordanti al prossimo spostamento, la funzione γ' garantisce anche che lo stesso carattere del pattern non combacerà con il carattere discordante del testo. Si mostri come calcolare la funzione γ' efficientemente.

Problemi

34-1 ***Corrispondenza tra stringhe basata sui fattori di ripetizione***

Si denoti con y^i la concatenazione della stringa y con sé stessa i volte. Per esempio, $(ab)^3$ = `ababab`. Si dice che una stringa $x \in \Sigma^*$ ha fattore di ripetizione r se $x = y^r$, per qualche

stringa $y \in \Sigma^*$ e qualche $r > 0$. Si denoti con $\rho(x)$ il più grande r tale che x abbia fattore di ripetizione r.

a. Si dia un algoritmo efficiente che prenda come input un pattern $P[1 .. m]$ e calcoli $\rho(P_i)$ per $i = 1, 2, \ldots, m$. Qual è il tempo di esecuzione dell'algoritmo?

b. Per qualunque pattern $P[1 .. m]$, sia $\rho^*(P)$ definito come $\max_{1 \le i \le m} \rho(P_i)$. Si dimostri che se il pattern P è scelto casualmente dall'insieme di tutte le stringhe binarie di lunghezza m, allora il valore medio di $\rho^*(P)$ è $O(1)$.

c. Si deduca che il seguente algoritmo per la corrispondenza tra stringhe trova correttamente tutte le occorrenze del pattern P in un testo $T[1 .. n]$ in tempo $O(\rho^*(P)n + m)$.

```
REPETITION-MATCHER(P,T)
m ← length[P]
n ← length[T]
k ← 1 + ρ*(P)
q ← 0
s ← 0
while s ≤ n − m
do   if T[s + q + 1] = P [q + 1]
        then q ← q + 1
                if q = m
                   then stampa "Il pattern appare con spostamento" s
     if q = m oppure T[s + q + 1] ≠ P [q + 1]
        then  s ← s + max(1, ⌈q/k⌉)
              q ← 0
```

Questo algoritmo è dovuto a Galil e Seiferas. Estendendo notevolmente queste idee, essi hanno ottenuto un algoritmo per la corrispondenza tra stringhe di tempo lineare che usa solo memoria $O(1)$ oltre a quella richiesta per P e T.

34-2 *Corrispondenza tra stringhe su un calcolatore parallelo*

Si consideri il problema della corrispondenza tra stringhe su un calcolatore parallelo. Si assuma che per un dato pattern si abbia un automa di corrispondenza tra stringhe M con insieme di stati Q. Sia ϕ la funzione di stato finale per M. Si supponga che il testo di input sia $T[1 .. n]$. Si desidera calcolare $\phi(T_i)$ per $i = 1, 2, \ldots, n$; cioè si desidera calcolare lo stato finale di ogni prefisso. La strategia è di usare il calcolo parallelo dei prefissi descritto al paragrafo 30.1.2.

Per qualunque stringa x di input, si definisce la funzione $\delta_x : Q \to Q$ tale che se M comincia dallo stato q e legge l'input x, allora M termina nello stato $\delta_x(q)$.

a. Si dimostri che $\delta_x + \delta_y = \delta_{xy}$, dove $\circ$ denota la composizione funzionale: $(\delta_y + \delta_x)(q) = \delta_y(\delta_x(q))$.

b. Si deduca che + è un'operazione associativa.

c. Si deduca che δ_{xy} può essere calcolata dalla rappresentazione tabellare di δ_x e δ_y in tempo $O(1)$ su una PRAM CREW. Si analizzi quanti processori sono necessari in termini di $|Q|$.

d. Si dimostri che $\phi(T_i) = \delta_{T_i}(q_0)$, dove q_0 è lo stato iniziale di M.

e. Si mostri come trovare tutte le occorrenze di un pattern in un testo di lunghezza n in tempo $O(\lg n)$ su una PRAM CREW. Si assuma che il pattern sia fornito nella forma del relativo automa di corrispondenza tra stringhe.

Note al capitolo

La relazione tra il problema della corrispondenza tra stringhe e la teoria degli automi finiti è discussa da Aho, Hopcroft e Ullman [4]. L'algoritmo di Knuth-Morris-Pratt [125] è stato inventato indipendentemente da Knuth e Pratt e da Morris, i quali però pubblicarono insieme i risultati del loro lavoro. L'algoritmo Rabin-Karp è stato proposto da Rabin e Karp [117] e l'algoritmo Boyer-Moore è dovuto a Boyer e Moore [32]. Galil e Seiferas [78] forniscono un interessante algoritmo deterministico di corrispondenza tra stringhe di tempo lineare che usa solo spazio $O(1)$ oltre a quello richiesto per memorizzare il pattern e il testo.

Geometria computazionale

35

La geometria computazionale è la branca dell'informatica che studia algoritmi per risolvere problemi geometrici. Essa trova applicazioni nell'ingegneria e nella matematica moderne, nei campi, tra gli altri, di grafica, robotica, cibernetica, progettazione di circuiti integrati su grandissima scala, progettazione assistita dal calcolatore, e statistica. L'input di un problema di geometria computazionale è tipicamente una descrizione di un insieme di oggetti geometrici, per esempio un insieme di punti, o un insieme di segmenti, o i vertici di un poligono in senso antiorario. L'output è spesso una risposta a una domanda sugli oggetti, per esempio se qualche linea si interseca con altre, o un nuovo oggetto geometrico, come per esempio l'inviluppo convesso di un insieme di punti (il più piccolo poligono convesso che racchiude tale insieme).

In questo capitolo, si vedranno alcuni algoritmi di geometria computazionale su due dimensioni, cioè nel piano. Ogni oggetto in input è rappresentato come un insieme di punti $\{p_i\}$, dove ogni $p_i = (x_i, y_i)$ e $x_i, y_i \in \mathbf{R}$. Per esempio, un poligono P di n vertici è rappresentato dalla sequenza $\langle p_0, p_1, p_2, \ldots, p_{n-1} \rangle$ dei suoi vertici presi nell'ordine in cui compaiono sul perimetro di P. La geometria computazionale può operare in tre dimensioni, o anche in spazi con un maggior numero di dimensioni, ma tali problemi e le loro soluzioni possono essere molto difficili da visualizzare. Anche in due dimensioni però, si può vedere una buona serie di tecniche di geometria computazionale.

Il paragrafo 35.1 mostra come rispondere a semplici domande su segmenti: se un segmento segue in senso orario o antiorario un altro con cui condivide un estremo; in quale senso girare quando si attraversano due segmenti consecutivi, e verificare se due segmenti si intersecano. Il paragrafo 35.2 presenta una tecnica chiamata "rastrellamento" utilizzata per sviluppare un algoritmo in tempo $O(n \lg n)$ che determina se vi sono intersezioni tra un insieme di n segmenti. Il paragrafo 35.3 fornisce due algoritmi di "rastrellamento rotazionale" che calcolano l'inviluppo convesso di un insieme di n punti, cioè il più piccolo poligono convesso che contiene tali punti: l'algoritmo di Graham, che impiega tempo $O(n \lg n)$, e l'algoritmo di Jarvis, che richiede tempo $O(nh)$, dove h è il numero di vertici dell'inviluppo convesso. Infine, al paragrafo 35.4 si fornisce un algoritmo divide-et-impera in tempo $O(n \lg n)$ per trovare la coppia di punti più vicini in un insieme di n punti sul piano.

35.1 Proprietà dei segmenti

Alcuni algoritmi di geometria computazionale di questo capitolo forniscono risposte a domande sulle proprietà dei segmenti. Una ***combinazione convessa*** di due punti distinti $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ è un qualunque punto $p_3 = (x_3, y_3)$ tale che per qualche α nell'intervallo

$[0, 1]$ si ha $x_3 = \alpha x_1 + (1-\alpha)x_2$ e $y_3 = \alpha y_1 + (1-\alpha)y_2$. Si scrive anche che $p_3 = \alpha p_1 + (1-\alpha)p_2$. **Intuitivamente**, p_3 è un qualunque punto sulla retta che passa per p_1 e p_2 e può coincidere con p_1 o p_2, oppure essere compreso tra i due punti. Dati due punti distinti p_1 e p_2, il ***segmento*** $\overline{p_1p_2}$ è l'insieme di combinazioni convesse di p_1 e p_2. p_1 e p_2 sono detti ***estremi*** del segmento $\overline{p_1p_2}$. Talvolta l'ordinamento tra p_1 e p_2 è importante e si parla di ***segmento orientato*** $\overrightarrow{p_1p_2}$. Se p_1 è l'origine (0, 0), allora si può trattare il segmento orientato $\overrightarrow{p_1p_2}$ come il ***vettore*** p_2.

In questo paragrafo, si analizzeranno le seguenti questioni:

1. Dati due segmenti orientati $\overrightarrow{p_0p_1}$ e $\overrightarrow{p_0p_2}$, $\overrightarrow{p_0p_1}$ segue in senso orario $\overrightarrow{p_0p_2}$ rispetto all'estremo comune p_0?
2. Dati due segmenti $\overline{p_1p_2}$ e $\overline{p_3p_4}$, se si attraversano $\overrightarrow{p_1p_2}$ e $\overrightarrow{p_3p_4}$, si effettua una svolta a sinistra nel punto p_2?
3. I segmenti $\overline{p_1p_2}$ e $\overline{p_3p_4}$ si intersecano?

Non vi sono restrizioni sui punti dati.

Si può rispondere ad ogni domanda in tempo $O(1)$, il che non dovrebbe sorprendere poiché la grandezza dell'input di ogni domanda è $O(1)$. Inoltre i metodi impiegati usano addizioni, sottrazioni, moltiplicazioni e confronti. Non c'è necessità né di divisione né di funzioni trigonometriche, operazioni entrambe computazionalmente costose e sottoposte a errori di arrotondamento. Per esempio, il metodo "banale" per determinare se due segmenti si intersecano calcola l'equazione della retta nella forma $y = mx + b$ per ogni segmento (m è l'inclinazione e b è l'intersezione con l'asse delle y), trova il punto di intersezione delle rette, e controlla se questo punto si trova su entrambi i segmenti (usa la divisione per trovare il punto di intersezione). Sui calcolatori reali, quando i segmenti sono quasi paralleli, questo metodo è molto sensibile alla precisione dell'operazione di divisione. Il metodo in questo paragrafo, che evita la divisione, è molto più accurato.

Prodotti incrociati

Il calcolo di prodotti incrociati è alla base degli algoritmi per segmenti che saranno presentati. Si considerino i vettori p_1 e p_2, mostrati nella figura 35.1(a). Il ***prodotto incrociato*** $p_1 \times p_2$ può essere interpretato come la superficie del parallelogramma formato dai punti (0, 0), p_1, p_2, e $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. Una definizione equivalente, ma più utile, descrive il prodotto incrociato come il determinante di una matrice[1]:

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1y_2 - x_2y_1 \\ &= -p_2 \times p_1 . \end{aligned}$$

Se $p_1 \times p_2$ è positivo, allora p_1 è disposto in senso orario rispetto a p_2 relativamente all'origine (0,0); se il prodotto incrociato è negativo, allora p_1 è in senso antiorario rispetto a p_2. La figura 35.1(b) mostra le regioni oraria e antioraria rispetto a un vettore p. Una condizione limite si verifica se il prodotto è zero; in questo caso i vettori sono ***collineari***, orientati nella stessa o nell'opposta direzione.

[1] Effettivamente, il prodotto incrociato è un concetto tridimensionale. È un vettore perpendicolare sia a p_1 che a p_2 secondo la "regola della mano destra", il cui modulo è $|x_1y_2 - x_2y_1|$. In questo capitolo, però, si considera più conveniente trattare il prodotto incrociato semplicemente come il valore $x_1y_2 - x_2y_1$.

Figura 35.1 (a) *Il prodotto incrociato dei vettori p_1 e p_2 è la superficie del parallelogramma.* (b) *La regione in grigio chiaro contiene vettori che seguono p in senso orario. La regione in grigio scuro contiene vettori che seguono p in senso antiorario.*

Per determinare se un segmento orientato $\overrightarrow{p_0p_1}$ segue un segmento orientato $\overrightarrow{p_0p_2}$ in senso orario relativamente al loro estremo in comune p_0, si effettua semplicemente una traslazione in modo che p_0 diventi l'origine. Se si denota con $p_1 - p_0$ il vettore $p_1' = (x_1', y_1')$, dove $x_1' = x_1 - x_0$ e $y_1' = y_1 - y_0$ e si definisce $p_2 - p_0$ analogamente, si può allora calcolare il prodotto incrociato

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) .$$

Se questo prodotto incrociato è positivo, allora $\overrightarrow{p_0p_1}$ segue $\overrightarrow{p_0p_2}$ in senso orario, se è negativo esso segue $\overrightarrow{p_0p_2}$ in senso antiorario.

Determinazione della svolta a sinistra o a destra tra segmenti consecutivi

Il prossimo problema è decidere se due segmenti consecutivi $\overline{p_0p_2}$ e $\overline{p_1p_2}$ effettuano una svolta a sinistra o a destra nel punto p_1. Equivalentemente, si vuole un metodo per determinare in quale senso gira un dato angolo $\angle p_0p_1p_2$. I prodotti incrociati consentono di rispondere a questa domanda senza calcolare l'angolo. Come mostrato nella figura 35.2, si controlla semplicemente se il segmento orientato $\overrightarrow{p_0p_2}$ segue in senso orario o antiorario il segmento orientato $\overrightarrow{p_0p_1}$. Per far ciò si calcola il prodotto incrociato $(p_2 - p_0) \times (p_1 - p_0)$. Se il prodotto incrociato è negativo, allora $\overrightarrow{p_0p_2}$ segue in senso antiorario $\overrightarrow{p_0p_1}$, pertanto si effettua una svolta a sinistra di p_1. Un prodotto incrociato positivo indica un orientamento orario e una svolta a destra. Un prodotto incrociato uguale a 0 significa che i punti p_0, p_1 e p_2 sono collineari.

Verifica dell'intersezione tra due segmenti

Per determinare se due segmenti si intersecano si usa un processo a due stadi. Il primo stadio è il ***rifiuto immediato***: i segmenti non possono intersecarsi se i loro rettangoli delimitanti non si intersecano. Il ***rettangolo delimitante*** di una figura geometrica è il più piccolo rettangolo che contiene la figura e i cui lati sono paralleli agli assi x e y. Il rettangolo delimitante del segmento $\overline{p_1p_2}$ è rappresentato dal rettangolo $\langle \hat{p}_1, \hat{p}_2 \rangle$ con vertice in basso a sinistra $\hat{p}_1 = (\hat{x}_1, \hat{y}_1)$ e vertice in alto a destra $\hat{p}_2 = (\hat{x}_2, \hat{y}_2)$, dove $\hat{x}_1 = \min(x_1, x_2)$, $\hat{y}_1 = \min(y_1, y_2)$, $\hat{x}_2 = \max(x_1, x_2)$, $\hat{y}_2 = \max(y_1, y_2)$. Due rettangoli, rappresentati dai vertici in basso a sinistra e in alto a destra

Figura 35.2 *Utilizzo del prodotto incrociato per determinare come i segmenti consecutivi $\overline{p_0p_1}$ e $\overline{p_1p_2}$ effettuano una svolta nel punto p_1. Si controlla se il segmento orientato $\overrightarrow{p_0p_1}$ segue in senso orario o antiorario il segmento orientato $\overrightarrow{p_0p_1}$.* (a) *Se è in senso antiorario, la svolta è a sinistra.* (b) *Se è in senso orario, la svolta è a destra.*

$(\hat{p}_1, \hat{p}_2)$ e $(\hat{p}_3, \hat{p}_4)$, si intersecano se e solo se la congiunzione

$$(\hat{x}_2 \geq \hat{x}_3) \wedge (\hat{x}_4 \geq \hat{x}_1) \wedge (\hat{y}_2 \geq \hat{y}_3) \wedge (\hat{y}_4 \geq \hat{y}_1)$$

è vera. I rettangoli devono intersecarsi su entrambe le dimensioni. I primi due confronti determinano se i rettangoli si intersecano in x; gli altri due confronti determinano se i rettangoli si intersecano in y.

Il secondo stadio per determinare se due segmenti si intersecano decide se ogni segmento "inforca" la retta che contiene l'altro. Un segmento $\overline{p_1p_2}$ ***inforca*** una retta se il punto p_1 si trova da un lato della retta e il punto p_2 si trova dall'altro lato. Se p_1 o p_2 si trovano sulla retta, allora si dice che il segmento inforca la retta. Due segmenti si intersecano se e solo se passano il controllo del rifiuto immediato ed ogni segmento inforca la retta contenente l'altro.

Si può usare il metodo dei prodotti incrociati per determinare se il segmento $\overline{p_3p_4}$ inforca la retta contenente i punti p_1 e p_2. L'idea, come mostrato nella figura 35.3(a) e (b), è di determinare se i segmenti orientati $\overrightarrow{p_1p_3}$ e $\overrightarrow{p_1p_4}$ hanno direzioni opposte rispetto a $\overrightarrow{p_1p_2}$. Se è così, allora il segmento inforca la retta. Si ricordi che è possibile determinare la direzione relativa con i prodotti incrociati, basta controllare se i segni dei prodotti $(p_3 - p_1) \times (p_2 - p_1)$ e $(p_4 - p_1) \times (p_2 - p_1)$ sono diversi. Una condizione limite si verifica se uno dei prodotti è uguale a 0. In questo caso uno tra p_3 e p_4 si trova sulla retta che contiene il segmento $\overline{p_1p_2}$. Due situazioni di questo tipo sono mostrate nella figura 34.3(c) e (d). Il caso in cui i due segmenti sono collineari ma non si intersecano, mostrato nella figura 35.3(e), è escluso dal controllo di rifiuto immediato. Una condizione limite si verifica se uno o entrambi i segmenti hanno lunghezza nulla, cioè se i due estremi coincidono. Se entrambi i segmenti hanno lunghezza zero, allora è sufficiente il controllo di rifiuto immediato. Se solo un segmento, per esempio $\overline{p_3p_4}$, ha lunghezza zero, allora i segmenti si intersecano se e solo se il prodotto incrociato $(p_3 - p_1) \times (p_2 - p_1)$ è zero.

Altre applicazioni dei prodotti incrociati

L'ultimo paragrafo di questo capitolo presenterà altri utilizzi dei prodotti incrociati. Al paragrafo 35.3, si dovrà ordinare un insieme di punti in base ai loro angoli polari rispetto a una data origine. Come richiesto dall'Esercizio 35.1-2, i prodotti incrociati possono essere usati per effettuare i confronti nella procedura di ordinamento. Al paragrafo 35.2, si useranno gli RB-alberi per mantenere l'ordinamento verticale di un insieme di segmenti che non si

Figura 35.3 *Il problema di determinare il segmento $\overline{p_3p_4}$ inforca la linea contenente il segmento $\overline{p_1p_2}$.* (a) *Se la inforca, allora i segni dei prodotti incrociati $(p_3 - p_1) \times (p_2 - p_1)$ e $(p_4 - p_1) \times (p_2 - p_1)$ sono diversi.* (b) *Se non la inforca, allora i segni dei prodotti incrociati sono uguali.* (c)-(d) *Casi limite in cui almeno uno dei prodotti incrociati è 0 è il segmento inforca la linea.* (e) *Un caso limite in cui i segmenti sono collineari ma non si intersecano. Entrambi i prodotti incrociati sono 0 ma essi non vengono calcolati dall'algoritmo poiché i segmenti non passano il controllo di rifiuto immediato – i loro rettangoli delimitanti non si intersecano.*

intersecano. Piuttosto che mantenere espliciti valori delle chiavi, nel codice per l'RB-albero si sostituirà ogni confronto tra chiavi con il calcolo dei prodotti incrociati, in modo da determinare quale tra due segmenti che si intersecano con una data retta verticale si trova sopra l'altro.

Esercizi

35.1-1 Si dimostri che se $p_1 \times p_2$ è positivo, allora il vettore p_1 segue in senso orario p_2 relativamente all'origine (0, 0) e che se questo prodotto incrociato è negativo, allora p_1 segue in senso antiorario p_2.

35.1-2 Si scriva lo pseudocodice per ordinare una sequenza $\langle p_1, p_2, \ldots, p_n \rangle$ di n punti in base ai loro angoli polari rispetto a un dato punto p_0 di origine. La procedura dovrebbe richiedere tempo $O(n \lg n)$: si usino i prodotti incrociati per confrontare gli angoli.

35.1-3 Si mostri come determinare in tempo $O(n^2 \lg n)$ se per qualche tripla di punti scelti in un insieme di n punti gli elementi della tripla sono collineari.

35.1-4 Il professor Amundsen propose il seguente metodo per determinare se una sequenza $\langle p_0, p_1, \ldots, p_{n-1} \rangle$ di n punti forma vertici consecutivi di un poligono convesso. (Si veda il paragrafo 16.4 per le definizioni che riguardano i poligoni.) Il risultato è "sì" se l'insieme $\{-p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, n-1\}$, dove l'addizione sugli indici in basso è eseguita modulo n, non contiene svolte sia a sinistra che a destra; altrimenti, il risultato è "no". Si mostri che malgrado questo metodo

impieghi tempo lineare, non sempre produce risultati corretti. Si modifichi il metodo in modo che produca sempre la risposta corretta in tempo lineare.

35.1-5 Dato un punto $p_0 = (x_0, y_0)$, il ***raggio orizzontale destro*** di p_0 è l'insieme di punti $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ e } y_i = y_0\}$, cioè l'insieme di punti che stanno a destra di p_0 compreso p_0 stesso. Si mostri come determinare in tempo $O(1)$ se un dato raggio orizzontale destro di p_0 interseca un segmento $\overline{p_1p_2}$ riducendo il problema a quello di determinare se due segmenti si intersecano.

35.1-6 Un modo per determinare se un punto p_0 si trova all'interno di un poligono P, non necessariamente convesso, è di considerare qualunque raggio di p_0 e controllare che il raggio intersechi il perimetro di P un numero dispari di volte, ma che p_0 stesso non si trovi sul perimetro. Si mostri come calcolare in tempo $\Theta(n)$ se un punto p_0 si trova all'interno di un poligono P di n vertici. (*Suggerimento:* si usi l'Esercizio 35.1-5. Assicurarsi che l'algoritmo sia corretto quando il raggio interseca il perimetro in un vertice e quando il raggio si sovrappone a un lato del poligono.)

35.1-7 Si mostri come calcolare in tempo $\Theta(n)$ la superficie dell'area di un poligono semplice di n vertici, ma non necessariamente convesso.

35.2 Verifica dell'intersezione di una coppia qualsiasi di segmenti

Questo paragrafo presenta un algoritmo per determinare se due segmenti qualsiasi appartenenti ad un insieme di segmenti si intersecano. L'algoritmo usa una tecnica conosciuta come "rastrellamento" che è comune a molti algoritmi di geometria computazionale. Inoltre, come richiesto dagli esercizi alla fine di questo paragrafo, questo algoritmo o semplici variazioni di esso, possono essere usati per risolvere altri problemi di geometria computazionale.

L'algoritmo impiega tempo $O(n \lg n)$, dove n è il numero di segmenti dati. Esso verifica solo se esiste qualche intersezione; non restituisce tutte le intersezioni. (Dall'Esercizio 35.2-1, l'algoritmo richiede tempo $\Omega(n^2)$ nel caso peggiore per trovare *tutte* le intersezioni in un insieme di n segmenti.)

Nel ***rastrellamento***, una immaginaria retta verticale, detta ***retta di rastrellamento***, passa attraverso l'insieme di oggetti geometrici dati, di solito da sinistra a destra. La dimensione dello spazio attraverso cui la retta di rastrellamento si muove, in questo caso la dimensione x, è trattata come una dimensione del tempo. Il rastrellamento fornisce un metodo per ordinare oggetti geometrici, di solito inserendoli in una struttura di dati dinamica e sfruttando le relazioni tra essi. L'algoritmo per l'intersezione tra segmenti di questo paragrafo considera tutti gli estremi dei segmenti nell'ordine da sinistra a destra e controlla se esiste un'intersezione ogni volta che incontra un vertice.

L'algoritmo per determinare se una coppia qualsiasi di segmenti, scelti tra n, si interseca, fa due ipotesi di semplificazione. In primo luogo, si assume che nessun segmento in input sia verticale. In secondo luogo si assume che non esistano tre segmenti che si intersecano in un solo punto. (L'Esercizio 35.2-8 richiede di descrivere una procedura che funzioni anche se queste ipotesi non valgono.) La rimozione di tali ipotesi di semplificazione e la gestione di casi limite costituiscono spesso la parte più difficile della programmazione di algoritmi di geometria computazionale e della dimostrazione della loro correttezza.

Figura 35.4 *L'ordinamento tra segmenti per varie linee di rastrellamento verticali.* (a) *Si ha* $a >_r c$, $a >_t b$, $b >_t c$, $a >_u c$ *e* $b >_u c$. *Il segmento* d *non è confrontabile con alcun segmento tra quelli mostrati.* (b) *Quando i segmenti* e *ed* f *si intersecano, i loro ordinamenti sono invertiti: si ha* $e >_v f$ *ma* $f >_w e$. *Qualsiasi retta di rastrellamento (per esempio* z*) che passa attraverso la regione grigia ha nel suo ordinamento totale* e *ed* f *consecutivi.*

Ordinamento di segmenti

Poiché si assume che non vi siano segmenti verticali, qualunque segmento di input che intersechi una data retta di rastrellamento verticale, la interseca in un solo punto. Si possono pertanto ordinare i segmenti che intersecano una retta di rastrellamento verticale rispetto alle coordinate y dei punti di intersezione.

Per essere più precisi si considerino due segmenti s_1 e s_2 che non si intersecano. Si dice che questi segmenti sono ***confrontabili*** in x se la retta di rastrellamento verticale con ascissa x li interseca entrambi. Si dice che s_1 è ***sopra*** s_2 in x, e si scrive $s_1 >_x s_2$, se s_1 e s_2 sono confrontabili in x e l'intersezione di s_1 con la retta di rastrellamento in x è più alta dell'intersezione di s_2 con la stessa retta di rastrellamento. Nella figura 35.4(a), per esempio, si hanno le relazioni $a >_r c$, $a >_t b$, $b >_t c$, $a >_u c$, e $b >_u c$. Il segmento d non è confrontabile con alcun altro segmento.

Dato un x qualsiasi, la relazione "$>_x$" impone un ordinamento totale (si veda il paragrafo 5.2) sui segmenti che intersecano la retta di rastrellamento in x. Però l'ordine può essere diverso per valori diversi di x, poiché i segmenti entrano ed escono dall'ordinamento. Un segmento entra nell'ordinamento quando la retta di rastrellamento si incontra con il suo estremo sinistro e ne esce quando viene incontrato il suo estremo destro.

Cosa accade quando la retta di rastrellamento passa attraverso l'intersezione di due segmenti? Come è mostrato nella figura 35.4(b), le loro posizioni nell'ordinamento totale sono invertite. Le linee di rastrellamento v e w sono rispettivamente alla sinistra e alla destra del punto di intersezione dei segmenti e ed f e si ha $e >_v f$ e $f >_w e$. Si noti che, poiché si assume che non vi siano tre segmenti che si intersecano nello stesso punto, deve esistere qualche retta di rastrellamento verticale x per cui le intersezioni dei segmenti e ed f sono *consecutive* nell'ordinamento totale $>_x$. Qualsiasi retta di rastrellamento che passi attraverso la regione in grigio della figura 35.4(b), come per esempio z, ha e ed f consecutivi nel suo ordinamento totale.

Spostamento della retta di rastrellamento

L'algoritmo di rastrellamento tipicamente gestisce due insiemi di dati:

1. Lo ***stato della retta di rastrellamento*** fornisce la relazione tra gli oggetti intersecati dalla retta di rastrellamento.

2. Il ***prospetto dei punti di arresto*** è una sequenza di ascisse, ordinate da sinistra a destra, che definisce la posizione di arresto della retta di rastrellamento. Ognuna di tali posizioni è detta punto di arresto. I cambiamenti dello stato della retta di rastrellamento si verificano solo per i punti di arresto.

Per alcuni algoritmi (l'algoritmo richiesto dall'Esercizio 35.2-7, per esempio), il prospetto dei punti di arresto è determinato dinamicamente man mano che l'algoritmo viene eseguito. L'algoritmo presentato nel seguito, però, determina i punti di arresto staticamente, basandosi esclusivamente sulle semplici proprietà dei dati in input. In particolare ogni estremo di segmento è un punto di arresto. Si ordinano gli estremi dei segmenti incrementando l'ascissa e procedendo da sinistra a destra. Si inserisce un segmento nello stato della retta di rastrellamento quando viene incontrato il suo estremo sinistro e si cancella dallo stato della retta di rastrellamento quando viene incontrato il suo estremo destro. Quando due segmenti diventano consecutivi nell'ordinamento totale, si controlla se si intersecano.

Lo stato della retta di rastrellamento è un ordinamento totale T per cui sono richieste le seguenti operazioni:

- INSERT(T,s): inserisce il segmento s in T.
- DELETE(T,s): cancella il segmento s da T.
- ABOVE(T,s): restituisce il segmento immediatamente sopra il segmento s in T.
- BELOW(T,s): restituisce il segmento immediatamente sotto il segmento s in T.

Se vi sono n segmenti di input, si può eseguire ognuna delle operazioni precedenti in tempo $O(\lg n)$ usando gli RB-alberi. Si ricorda che le operazioni sugli RB-alberi al Capitolo 14 prevedono il confronto tra chiavi. Si possono sostituire i confronti tra chiavi con confronti tra prodotti incrociati che determinano l'ordinamento relativo di due segmenti (si veda l'Esercizio 35.2-2).

Algoritmo per l'intersezione tra segmenti

Il seguente algoritmo prende in input un insieme S di n segmenti, restituendo il valore booleano TRUE se una coppia qualsiasi di segmenti in S si interseca, FALSE altrimenti. L'ordinamento totale T è realizzato con un RB-albero.

```
ANY-SEGMENTS-INTERSECT(S)
1  T ← ∅
2  ordina gli estremi dei segmenti in S da sinistra a destra
       e risolve situazioni ambigue inserendo prima i punti con ordinata più bassa
3  for ogni punto p nella lista ordinata di estremi
4      do  if p è l'estremo sinistro di un segmento s
5          then  INSERT(T,s)
6               if (ABOVE(T,s) esiste ed interseca s)
                    oppure (BELOW(T,s) esiste ed interseca s)
7               then return TRUE
8      if p è l'estremo destro di un segmento s
9          then  if ABOVE(T,s) e BELOW(T,s) esistono entrambi
                    e ABOVE(T,s) interseca (BELOW(T,s))
10              then return TRUE
11              DELETE(T,s)
12 return FALSE
```

Figura 35.5 *L'esecuzione di* ANY-SEGMENTS-INTERSECT. *Ogni retta tratteggiata è la retta di rastrellamento verticale in quel dato punto di arresto e l'ordinamento dei nomi dei segmenti sotto ogni retta di rastrellamento è l'ordinamento totale T alla fine del ciclo* **for** *in cui il corrispondente punto di arresto è analizzato. L'intersezione dei segmenti d e b viene trovata quando è cancellato il segmento c.*

La figura 35.5 illustra l'esecuzione dell'algoritmo. Alla linea 1 si inizializza a vuoto l'ordinamento totale. Alla linea 2 si determina il prospetto dei punti di arresto ordinando i $2n$ estremi dei segmenti da sinistra a destra e risolvendo le situazioni ambigue inserendo prima i punti con ordinata più bassa. Si noti che la linea 2 può essere eseguita ordinando lessicograficamente gli estremi rispetto a (x,y).

Ogni iterazione del ciclo **for** alle linee 3-11 considera un punto di arresto p. Se p è l'estremo sinistro di un segmento s, alla linea 5 si aggiunge s all'ordinamento totale e alle linee 6-7 è restituito TRUE se s interseca uno dei due segmenti ad esso consecutivi rispetto all'ordinamento totale definito dalla retta di rastrellamento che passa per p. (Una condizione limite si verifica se p si trova su un altro segmento s'. In questo caso si richiede soltanto che s e s' siano consecutivi in T.) Se p è l'estremo destro di un segmento s, allora s deve essere cancellato dall'ordinamento totale. Alle linee 9-10 è restituito TRUE se vi è un'intersezione tra i segmenti che sono accanto a s nell'ordinamento totale definito dalla retta di rastrellamento che passa per p; questi segmenti diventeranno consecutivi nell'ordinamento totale quando s sarà cancellato. Se questi segmenti non si intersecano, la linea 11 cancella il segmento s

dall'ordinamento totale. Infine, se non è trovata alcuna intersezione nella verifica di tutti i $2n$ punti di arresto, alla linea 12 è restituito FALSE.

Correttezza

Il seguente Teorema mostra che ANY-SEGMENT-INTERSECT è corretta.

Teorema 35.1

La chiamata ANY-SEGMENT-INTERSECT(S) restituisce TRUE se e solo se vi è un'intersezione tra una coppia qualsiasi di segmenti in S.

Dimostrazione. La procedura può essere scorretta solo se restituisce TRUE quando non vi sono intersezioni o restituisce FALSE quando vi è almeno un'intersezione. Il primo caso non può verificarsi perché ANY-SEGMENT-INTERSECT restituisce TRUE solo se trova un'intersezione tra due dei segmenti di input.

Per mostrare che l'altro caso non può verificarsi, si supponga per assurdo che vi sia almeno un'intersezione e che ANY-SEGMENT-INTERSECT restituisca FALSE. Sia p il punto di intersezione più a sinistra, scegliendo in caso di ambiguità quello con l'ordinata più bassa, e siano a e b i segmenti che si intersecano in p. Poiché nessuna intersezione si verifica alla sinistra di p, l'ordine dato da T è corretto per tutti i punti alla sinistra di p. Poiché non ci sono tre segmenti che si intersecano nello stesso punto, esiste una retta di rastrellamento z per cui a e b diventano consecutivi nell'ordinamento totale[2]. Inoltre, z è alla sinistra di p oppure attraversa p. Esiste un estremo q sulla retta di rastrellamento z che è un punto di arresto per cui a e b diventano consecutivi nell'ordinamento totale. Se p è sulla retta di rastrellamento z allora $q = p$. Se p non è sulla retta di rastrellamento z, allora q è alla sinistra di p. In ogni caso l'ordinamento dato da T appena prima che q venga analizzato è corretto. (Ci si basa sul fatto che p sia il più basso dei punti di intersezione a sinistra. A causa dell'ordinamento lessicografico con cui sono analizzati i punti di arresto, anche se p è sulla retta di rastrellamento z e vi è un altro punto di intersezione p' su z, il punto di arresto $q = p$ è analizzato prima che l'altro punto di intersezione p' possa interferire sull'ordinamento totale T.) Vi sono solo due possibilità per le azioni intraprese per un dato punto di arresto q:

1. In T è inserito a oppure b e l'altro segmento è sopra o sotto quello inserito nell'ordinamento totale. Le linee 4-7 controllano questo caso.
2. I segmenti a e b sono già in T e un segmento tra essi nell'ordinamento totale viene cancellato, rendendo a e b consecutivi. Le linee 8-11 controllano questo caso.

In entrambi i casi, viene trovata l'intersezione p contraddicendo l'ipotesi che la procedura restituisca FALSE. ■

[2] Se si consente l'intersezione nello stesso punto di tre segmenti, vi potrebbe essere un segmento c che interseca sia a che b nel punto p. Cioè, si potrebbe avere $a <_w c$ e $c <_w b$ per tutte le linee w di rastrellamento alla sinistra di p per cui $a <_w b$.

Tempo di esecuzione

Se vi sono n segmenti nell'insieme S, allora ANY-SEGMENT-INTERSECT impiega tempo $O(n \lg n)$. La linea 1 richiede tempo $O(1)$. La linea 2 richiede tempo $O(n \lg n)$, usando il merge sort o lo heap sort. Poiché vi sono $2n$ punti di arresto, il ciclo **for** alle linee 3-11 si ripete al più $2n$ volte. Ogni iterazione richiede tempo $O(\lg n)$, poiché ogni operazione su un RB-albero richiede tempo $O(\lg n)$ e, usando il metodo del paragrafo 35.1, ogni controllo di intersezione richiede tempo $O(1)$. Il tempo totale è pertanto $O(n \lg n)$.

Esercizi

35.2-1 Si mostri che vi possono essere $\Theta(n^2)$ intersezioni in un insieme di n segmenti.

35.2-2 Dati due segmenti a e b che non si intersecano e che sono confrontabili in x, si mostri come usare i prodotti incrociati per determinare in tempo $O(1)$ quale vale tra $a >_x b$ e $b >_x a$.

35.2-3 Il professor Maginot suggerisce di modificare ANY-SEGMENT-INTERSECT in modo che invece di terminare dopo aver trovato un'intersezione, stampi i segmenti che si intersecano e continui l'iterazione del ciclo **for**. Il professore chiama la procedura risultante PRINT-INTERSECTING-SEGMENT e afferma che stampa tutte le intersezioni, da sinistra a destra, man mano che si verificano nell'insieme dei segmenti. Si mostri che la procedura proposta dal professore fallisce in due casi, dando, come contro esempi, un insieme di segmenti per cui la prima intersezione trovata da PRINT-INTERSECTING-SEGMENT non è quella più a sinistra e un insieme di segmenti per cui PRINT-INTERSECTING-SEGMENT non riesce a trovare tutte le intersezioni.

35.2-4 Si dia un algoritmo di tempo $O(n \lg n)$ per determinare se un poligono di n vertici è semplice.

35.2-5 Si dia un algoritmo di tempo $O(n \lg n)$ per determinare se due poligoni semplici con un totale di n vertici si intersecano.

35.2-6 Un *cerchio* è formato da una circonferenza e dal suo interno ed è rappresentato dal suo punto di centro e dal raggio. Due cerchi si intersecano se hanno qualche punto in comune. Si dia un algoritmo di tempo $O(n \lg n)$ per determinare se due cerchi presi in un insieme di n cerchi si intersecano.

35.2-7 Dato un insieme di n segmenti contenente in totale k intersezioni, si mostri come restituire tutte le k intersezioni in tempo $O((n + k)\lg n)$.

35.2-8 Si mostri come realizzare le procedure per RB-alberi così che ANY-SEGMENT-INTERSECT funzioni correttamente anche se qualche segmento è verticale o se tre segmenti o più si intersecano nello stesso punto. Si dimostri che la realizzazione è corretta.

35.3 Calcolo dell'inviluppo convesso

L'***inviluppo convesso*** di un insieme Q di punti è il più piccolo poligono convesso P per cui ogni punto di Q si trova sul perimetro o all' interno di P. Si denoti l'inviluppo convesso di Q con CH(Q) (CH sta per "convex hull", il termine inglese per inviluppo convesso). Intuitivamente si può pensare che ogni punto di Q sia un chiodo sporgente da un asse. L'inviluppo convesso è allora la forma tracciata da un piccolo elastico che circonda tutti i chiodi. La figura 35.6 mostra un insieme di punti e il suo inviluppo convesso.

In questo paragrafo, saranno presentati due algoritmi che calcolano l'inviluppo convesso di un insieme di n punti. Entrambi gli algoritmi restituiscono i vertici dell'inviluppo convesso seguendo il senso antiorario. Il primo, conosciuto come l'algoritmo di Graham, impiega tempo $O(n \lg n)$. Il secondo, chiamato algoritmo di Jarvis, impiega tempo $O(nh)$, dove h è il numero di vertici dell'inviluppo convesso. Come si può vedere dalla figura 35.6, ogni vertice di CH(Q) è un punto di Q. Entrambi gli algoritmi impiegano questa proprietà, decidendo quali vertici di Q mantenere come vertici dell'inviluppo convesso e quali ignorare.

In realtà esiste un certo numero di metodi per calcolare gli inviluppi convessi in tempo $O(n \lg n)$. Entrambi gli algoritmi di Graham e Jarvis usano una tecnica chiamata "rastrellamento rotazionale" che analizza i vertici seguendo l'ordine degli angoli polari che essi formano con un vertice di riferimento. Si segnalano i seguenti metodi.

- Nel ***metodo incrementale***, i punti sono ordinati da sinistra a destra, portando a una sequenza $\langle p_1, p_2, \ldots, p_n \rangle$. All'$i$-esimo stadio, l'inviluppo convesso CH($\{p_1, p_2, \ldots, p_{i-1}\}$) degli $i-1$ punti più a sinistra è aggiornato in base all'i-esimo punto da sinistra, formando così CH($\{p_1, p_2, \ldots, p_i\}$). Come l'Esercizio 35.3-6 richiede di mostrare, questo metodo può essere realizzato in modo da richiedere un tempo totale $O(n \lg n)$.
- Nel ***metodo divide-et-impera***, l'insieme di n punti è diviso, in tempo $\Theta(n)$, in due sottoinsiemi, uno degli $\lceil n/2 \rceil$ punti più a sinistra e uno degli $\lfloor n/2 \rfloor$ punti più a destra, gli inviluppi convessi dei sottoinsiemi sono calcolati ricorsivamente e quindi viene usato un metodo astuto per combinare gli inviluppi in tempo $O(n)$.
- Il ***metodo prune-and-search*** è simile all'algoritmo del paragrafo 10.3 per il calcolo del mediano che richiede tempo lineare nel caso peggiore. Trova la porzione superiore (o "catena superiore") dell'inviluppo convesso eliminando ripetutamente una frazione costante di punti finché rimane solo la catena superiore dell'inviluppo convesso. Fa poi la stessa cosa con la catena inferiore. Questo metodo è asintoticamente il più veloce: se l'inviluppo convesso contiene h vertici, esso richiede solo tempo $O(n \lg h)$.

Il calcolo dell'inviluppo convesso di un insieme di punti è un problema interessante di per sé. Inoltre, molti algoritmi che risolvono altri problemi di geometria computazionale cominciano con il calcolo dell'inviluppo convesso. Si consideri, per esempio il ***problema della coppia di punti più distanti*** in due dimensioni: dato un insieme di n punti del piano si desidera trovare due punti la cui distanza relativa sia massima. Come l'Esercizio 35.3-3 richiede di dimostrare, questi due punti devono essere vertici dell'inviluppo convesso. Sebbene non si voglia dimostrarlo qui, la coppia di vertici più distanti di un poligono convesso di n vertici può essere trovata in tempo $O(n)$. Pertanto, calcolando l'inviluppo convesso degli n punti in tempo $O(n \lg n)$ e trovando quindi la coppia di vertici più lontani del risultante poligono convesso, si può trovare la coppia di punti più distanti in un insieme di n punti in tempo $O(n \lg n)$.

Figura 35.6 *Un insieme Q di punti e il suo inviluppo convesso* CH(Q) *in grigio.*

Algoritmo di Graham

L'***algoritmo di Graham*** risolve il problema dell'inviluppo convesso mantenendo una pila S di punti candidati. Ogni punto dell'insieme Q è inserito una sola volta sulla pila e i punti che non sono vertici di CH(Q) sono eventualmente estratti dalla pila. Quando l'algoritmo termina, la pila S contiene esattamente i vertici di CH(Q) in ordine antiorario rispetto a come appaiono sul perimetro.

La procedura GRAHAM-SCAN prende come input un insieme Q di punti, dove $|Q| \geq 3$. Chiama la funzione TOP(S) che restituisce il punto in cima alla pila senza modificare S e NEXT-TO-TOP(S) che restituisce il punto sotto la cima della pila S senza modificare S. Come si dimostrerà tra poco, la pila S restituita da GRAHAM-SCAN contiene dal fondo alla cima esattamente i vertici di CH(Q) in ordine antiorario.

```
GRAHAM-SCAN(Q)
sia p0 il punto di Q con la minima ordinata, o il più a sinistra di tali punti in caso di parità
siano ⟨p1, p2, ..., pm⟩ i rimanenti punti di Q,
        ordinati rispetto all'angolo polare in ordine antiorario attorno a p0
        (se vi sono più punti con lo stesso angolo, rimuoverli tutti tranne
        quello che è più distante da p0)
top[S] ← 0
PUSH(p0, S)
PUSH(p1, S)
PUSH(p2, S)
for i ← 3 to m
    do    while l'angolo formato dai punti NEXT-TO-TOP(S),
                  TOP(S) e pi non provoca una svolta a sinistra
              do POP(S)
          PUSH(pi, S)
return S
```

Figura 35.7 *L'esecuzione di* GRAHAM-SCAN *sull'insieme Q della figura* 35.6. *L'inviluppo convesso corrente contenuto nella pila S è mostrato in grigio ad ogni passo.* (a) *Gli angoli polari ordinati di* $\langle p_1, p_2, \ldots, p_{12}\rangle$ *relativi a* p_0 *e la pila iniziale contenente* p_0, p_1 *e* p_2. (b)-(k) *La pila S dopo ogni iterazione del ciclo* **for** *delle linee* 7-10. *Le linee tratteggiate mostrano le svolte non a sinistra che causano l'estrazione dei punti dalla pila. Nella parte* (h), *per esempio, la svolta a destra nell'angolo* $\angle p_7 p_8 p_9$ *causa l'estrazione di* p_8 *e quindi la svolta a destra nell'angolo* $\angle p_6 p_7 p_9$ *causa l'estrazione di* p_7. (l) *L'inviluppo convesso restituito dalla procedura, che corrisponde a quello della figura* 35.6.

La figura 35.7 illustra l'esecuzione di GRAHAM-SCAN. Alla linea 1 si sceglie il punto p_0 come il punto con la minima ordinata, scegliendo il più a sinistra di tali punti in caso di parità. Poiché

Figura 35.7 *Continua.*

non vi è alcun punto di Q che sia sotto p_0 e qualunque altro punto con la stessa ordinata si trova alla sua destra, p_0 è un vertice di $CH(Q)$. Alla linea 2 si ordinano gli altri punti di Q rispetto a p_0 in base all'angolo polare usando lo stesso metodo, il confronto di prodotti incrociati, descritto all'Esercizio 35.1-2. Se due o più punti hanno lo stesso angolo polare rispetto a p_0, tutti tranne il più lontano di tali punti sono combinazioni convesse di p_0 e il punto più lontano, quindi possono essere completamente esclusi dalla trattazione. Si denoti con m il numero di

punti restanti diversi da p_0. L'angolo polare, misurato in radianti, di ogni punto in Q relativo a p_0 è nell'intervallo semiaperto $[0, \pi/2)$. Poiché gli angoli polari crescono in senso antiorario, i punti sono ordinati secondo l'ordine antiorario rispetto a p_0. Si denoti questa sequenza ordinata di punti con $\langle p_1, p_2, \ldots, p_m \rangle$. Si noti che i punti p_1 e p_m sono vertici di CH(Q) (si veda l'Esercizio 35.3-1). La figura 35.7(a) mostra i punti della figura 35.6, con gli angoli polari di $\langle p_1, p_2, \ldots, p_{12} \rangle$ ordinati rispetto a p_0.

Il resto della procedura usa la pila S. Le linee 3-6 inizializzano la pila perché contenga, dal basso verso l'alto, i primi tre punti p_0, p_1 e p_2. La figura 35.7(a) mostra la pila iniziale S. Il ciclo **for** alle linee 7-10 si ripete per ogni punto della sottosequenza $\langle p_3, p_4, \ldots, p_m \rangle$. Lo scopo è che dopo aver analizzato il punto p_i, la pila S contenga, dal basso verso l'alto, i vertici di CH($\{p_0, p_1, \ldots, p_i\}$) in ordine antiorario. Il ciclo **while** alle linee 8-9 rimuove dalla pila i punti che si è mostrato non essere vertici dell'inviluppo convesso. Quando si percorre l'inviluppo convesso in senso antiorario, si dovrebbe effettuare una svolta a sinistra per ogni vertice. Pertanto, ogni volta che il ciclo **while** trova un vertice in cui non si effettua una svolta a sinistra, il vertice è estratto dalla pila. (Il controllo della svolta non a sinistra, piuttosto che della svolta a destra, esclude la possibilità di un angolo retto in un vertice dell'inviluppo convesso risultante. Ciò è necessario poiché ogni vertice di un poligono convesso non deve essere una combinazione convessa di altri vertici del poligono.) Dopo aver estratto tutti i vertici che hanno svolte non a sinistra, quando ci si dirige verso il punto p_i, si inserisce p_i sulla pila. Le figure 35.7(b)-(k) mostrano lo stato della pila S dopo ogni iterazione del ciclo **for**. Infine, GRAHAM-SCAN restituisce la pila S alla linea 11. La figura 35.7(l) mostra il corrispondente inviluppo convesso.

Il seguente teorema dimostra formalmente la correttezza di GRAHAM-SCAN.

> ***Teorema 35.2 (Correttezza dell'algoritmo di Graham)***
>
> Se GRAHAM-SCAN è eseguito su un insieme Q di punti, dove $|Q| \geq 3$, allora un punto di Q si trova sulla pila S al termine della procedura se e solo se è un vertice di CH(Q).

Dimostrazione. Come notato sopra, un vertice che sia una combinazione convessa di p_0 e qualche altro vertice in Q non è un vertice di CH(Q). Un tale vertice non è incluso nella sequenza $\langle p_1, p_2, \ldots, p_m \rangle$, quindi non può mai apparire sulla pila.

Il punto cruciale della dimostrazione sta nelle due situazioni mostrate nella figura 35.8. La parte (a) descrive svolte non a sinistra, la parte (b) descrive svolte a sinistra.

Si mostra per prima cosa che ogni punto estratto dalla pila S non è un vertice di CH(Q). Si supponga che il punto p_j sia estratto dalla pila perché l'angolo $\angle p_k p_j p_i$ effettua una svolta non a sinistra, come mostrato nella figura 35.8(a). Poiché si scandiscono i punti seguendo gli angoli polari relativi al punto p_0 in senso antiorario, esiste un triangolo $\triangle p_0 p_i p_k$ con il punto p_j che si trova o all'interno del triangolo o sul segmento $\overline{p_i p_k}$. In entrambi i casi, il punto p_j non può essere un vertice di CH(Q).

Si mostra ora che al termine della procedura ogni punto sulla pila S è un vertice di CH(Q). Si comincia col dimostrare la seguente affermazione: la procedura GRAHAM-SCAN mantiene l'invariante che i punti sulla pila S formano sempre i vertici di un poligono convesso in ordine antiorario.

L'affermazione è vera subito dopo l'esecuzione della linea 6, poiché i punti p_0, p_1 e p_2 formano un poligono convesso. Si esamini ora come cambia la pila S durante l'esecuzione

Figura 35.8 *Le due situazioni su cui si basa la dimostrazione di correttezza dell'algoritmo* GRAHAM-SCAN. (a) *Illustrazione che un punto estratto dalla pila in* GRAHAM-SCAN *non è un vertice di* CH(Q). *Se il punto p_j è estratto dalla pila perché l'angolo $\angle p_k p_j p_i$ effettua una svolta non a sinistra, allora il triangolo in grigio $\triangle p_0 p_k p_i$ contiene il punto p_j. Il punto p_j quindi non è un vertice di* CH(Q). (b) *Se il punto p_i è inserito sulla pila, allora vi deve essere una svolta a sinistra nell'angolo $\angle p_k p_j p_i$. Poiché p_i segue p_j nell'ordinamento dei punti rispetto agli angoli polari e dato il modo in cui è stato scelto p_0, allora p_i deve essere nella regione in grigio. Se i punti nella pila formano un poligono convesso prima dell'inserimento, allora devono formare un poligono convesso anche dopo.*

di GRAHAM-SCAN. I punti sono o estratti o inseriti. Nel primo caso, ci si basa su una semplice proprietà geometrica: se un vertice è rimosso da un poligono convesso, il risultante poligono è convesso. Pertanto l'estrazione di un punto dalla pila mantiene l'invariante.

Prima di considerare il caso in cui un punto è inserito sulla pila, si esamina un'altra proprietà geometrica illustrata alle figure 35.9(a) e (b). Sia P un poligono convesso e si scelga un qualunque lato $\overline{p_r p_s}$ di P. Si consideri la regione limitata da $\overline{p_r p_s}$ e le estensioni dei due lati adiacenti. (A seconda degli angoli relativi dei lati adiacenti, la regione può essere o limitata, come la regione in grigio della parte (a), o illimitata, come la regione in grigio della parte (b).) Se si aggiunge qualche punto p_t in questa regione come nuovo vertice di P, con i lati $\overline{p_r p_t}$ e $\overline{p_t p_s}$ che sostituiscono $\overline{p_r p_s}$, il poligono risultante è convesso.

Si consideri ora un punto p_i che viene inserito nella pila S. In riferimento alla figura 35.8(b), sia p_j il vertice sulla cima di S appena prima che venga inserito p_i e sia p_k il predecessore di p_j su S. Si afferma che p_i deve cadere dentro la regione in grigio della figura 35.8(b) che corrisponde direttamente alla regione in grigio della figura 35.9. Poiché l'angolo $\angle p_k p_j p_i$ effettua una svolta a sinistra, p_i deve essere sul lato in grigio dell'estensione di $\overline{p_k p_j}$. Poiché p_i segue p_j nell'ordinamento rispetto all'angolo polare, p_i deve essere sul lato in grigio di $\overline{p_0 p_j}$. Inoltre, dato il modo in cui si sceglie p_0, il punto p_i deve essere sul lato in grigio dell'estensione di $\overline{p_0 p_1}$. Pertanto, p_i è nella regione in grigio e quindi, dopo che p_i è stato inserito nella pila S, i punti di S formano un poligono convesso. Con ciò si completa la dimostrazione dell'affermazione.

Alla fine di GRAHAM-SCAN, quindi, i punti di Q che sono sulla pila S formano i vertici di un poligono convesso. Si è mostrato che tutti i vertici di CH(Q) sono su S. Poiché S contiene solo vertici di Q e i suoi punti formano un poligono convesso, essi devono formare CH(Q). ■

Si mostra ora che il tempo di esecuzione di GRAHAM-SCAN è $O(n \lg n)$, dove $n = |Q|$. La linea 1 richiede tempo $\Theta(n)$. La linea 2 richiede tempo $O(n \lg n)$, usando merge sort o heapsort per ordinare gli angoli polari e il metodo dei prodotti incrociati del paragrafo 35.1 per confrontare gli angoli. (L'eliminazione di tutti i punti con lo stesso angolo polare tranne quello più lontano

Figura 35.9 *Aggiungendo un punto nella regione in grigio del poligono convesso P si ottiene un altro poligono convesso. La regione in grigio è limitata dal lato $\overline{p_r p_t}$ e dalle estensioni dei due lati ad esso adiacenti.* (a) *La regione in grigio è limitata.* (b) *La regione in grigio è illimitata.*

può essere realizzato con tempo totale $O(n)$.) Le linee 3-6 richiedono tempo $O(1)$. Dato che $m \le n-1$, il ciclo **for** alle linee 7-10 è eseguito al più $n-3$ volte. Poiché PUSH richiede tempo $O(1)$, ogni iterazione richiede tempo $O(1)$ senza considerare il tempo speso nel ciclo **while** alle linee 8-9; pertanto il tempo complessivo richiesto dal ciclo **for** è $O(n)$, escluso il tempo richiesto dal ciclo **while** interno.

Si usa il metodo degli aggregati per l'analisi ammortizzata per mostrare che il ciclo **while** richiede complessivamente tempo $O(n)$. Per $i = 0, 1, \ldots, m$, ogni punto p_i è inserito nella pila S esattamente una volta. Come nell'analisi della procedura MULTIPOP del paragrafo 18.1, si osserva che vi è al più un'operazione di POP per ogni operazione di PUSH. Almeno tre punti – p_0, p_1 e p_m – non sono mai estratti dalla pila, così che in effetti vengono eseguite in totale al più $m-2$ operazioni di POP. Ogni iterazione del ciclo **while** esegue una POP e così vi sono al più $m-2$ iterazioni del ciclo **while** complessivamente. Poiché il controllo alla linea 8 richiede tempo $O(1)$, ogni chiamata di POP richiede $O(1)$ e $m \le n-1$, il tempo totale richiesto dal ciclo **while** è $O(n)$. Pertanto, il tempo di esecuzione di GRAHAM-SCAN è $O(n \lg n)$.

Algoritmo di Jarvis

L'***algoritmo di Jarvis*** calcola l'inviluppo convesso di un insieme Q di punti attraverso una tecnica conosciuta come ***impacchettamento*** (in inglese ***package wrapping***). L'algoritmo impiega tempo $O(nh)$, dove h è il numero di vertici di CH(Q). Quando h è $o(\lg n)$, l'algoritmo di Jarvis è asintoticamente più veloce dell'algoritmo di Graham.

Intuitivamente, l'algoritmo di Jarvis simula l'avvolgimento dell'insieme Q con un foglio di carta. Si comincia con il fissare la fine del foglio sul punto più basso dell'insieme, cioè, lo stesso punto iniziale p_0 scelto dall'algoritmo di Graham. Questo punto è un vertice dell'inviluppo convesso. Quindi si sistema il foglio verso destra per tenderlo e poi verso l'alto finché tocca un punto. Anche questo punto deve essere un vertice dell'inviluppo convesso. Si continua in questo modo a stendere il foglio lungo l'insieme di vertici finché si torna al punto originale p_0.

Figura 35.10 *Il funzionamento dell'algoritmo di Jarvis. Il primo vertice scelto è il punto più in basso p_0. Il vertice successivo, p_1, è quello che forma l'angolo polare con p_0 più piccolo di qualunque altro punto. Quindi, p_2 forma l'angolo polare più piccolo con p_1. La catena destra arriva in alto fino al punto più alto p_3. Quindi è determinata la catena sinistra trovando gli angoli polari minimi rispetto al semiasse negativo delle x.*

Più formalmente, l'algoritmo di Jarvis costruisce una sequenza $H = \langle p_0, p_1, \ldots, p_{h-1} \rangle$ dei vertici di CH(Q). Si comincia con p_0; come mostra la figura 35.10, il vertice successivo p_1 dell'inviluppo convesso forma il minimo angolo polare con p_0. (In caso di parità, si sceglie il punto più lontano da p_0.) Analogamente, p_2 forma il minimo angolo polare con p_1 e così via. Quando si raggiunge il vertice più alto, per esempio p_k (la parità è risolta scegliendo il vertice più lontano), si è costruita, come mostra la figura 35.10, la ***catena destra*** di CH(Q). Per costruire la ***catena sinistra***, si comincia da p_k e si sceglie p_{k+1} come il punto che forma il minimo angolo polare con p_k, ma *rispetto al semiasse negativo delle x*. Si continua a costruire la catena sinistra considerando gli angoli polari rispetto al semiasse negativo delle x, finché si torna al vertice originale p_0.

Si potrebbe realizzare l'algoritmo di Jarvis con un rastrellamento concepito intorno all'inviluppo convesso, senza costruire separatamente le catene destra e sinistra. Tali realizzazioni tipicamente tengono traccia dell'angolo dell'ultimo lato scelto dell'inviluppo convesso e richiedono che la sequenza di angoli dei lati dell'inviluppo sia strettamente crescente (nell'intervallo tra 0 e 2π radianti). Il vantaggio di costruire catene separate è che non è necessario calcolare esplicitamente gli angoli; per confrontare gli angoli sono sufficienti le tecniche del paragrafo 35.1. Come mostra il paragrafo 10.1, si può calcolare il minimo di n valori in tempo $O(n)$ se ogni confronto richiede tempo $O(1)$. Pertanto, l'algoritmo di Jarvis richiede tempo $O(nh)$.

Esercizi

35.3-1 Si dimostri che nella procedura GRAHAM-SCAN, i punti p_1 e p_m devono essere vertici di CH(Q).

Figura 35.11 *La definizione di un poligono a forma di stella, da usare nell'Esercizio 35.3-4.* (a) *Un poligono a forma di stella. Il segmento dal punto p ad un qualsiasi punto q sul perimetro interseca il perimetro solo nel punto q.* (b) *Un poligono non a forma di stella. La regione in grigio a sinistra è l'ombra di q e la regione in grigio a destra è l'ombra di q'. Poiché queste regioni sono disgiunte, il nucleo è vuoto.*

35.3-2 Si consideri un modello di calcolo che offra l'addizione, il confronto e la moltiplicazione e per cui sia previsto un limite inferiore $\Omega(n \lg n)$ per l'ordinamento di n numeri. Si dimostri che in tale modello, $\Omega(n \lg n)$ è un limite inferiore del calcolo dei vertici ordinati dell'inviluppo convesso di un insieme di n punti.

35.3-3 Dato un insieme Q di punti, si dimostri che i due punti più lontani tra loro sono vertici di $CH(Q)$.

35.3-4 Dato un poligono P ed un punto q sul suo perimetro, l'***ombra*** di q è l'insieme dei punti r tale che il segmento $\overline{qr}$ è completamente sul perimetro o all'interno di P. Un poligono P è ***a forma di stella*** se esiste un punto p all'interno di P che è nell'ombra di ogni punto del perimetro di P. L'insieme di tutti i punti di questo tipo è chiamato il ***nucleo*** di P. (Si veda la figura 35.11.) Dato un poligono a forma di stella di n vertici, specificato dai suoi vertici in ordine antiorario, si mostri come calcolare $CH(P)$ in tempo $O(n)$.

35.3-5 Nel ***problema dell'inviluppo convesso in tempo reale***, l'insieme Q di n punti è dato fornendo un punto alla volta. Via via che ogni punto è fornito, si calcola l'inviluppo convesso dei punti disponibili. Ovviamente, si potrebbe eseguire l'algoritmo di Graham una volta per ogni punto, con un tempo di esecuzione complessivo $O(n^2 \lg n)$. Si mostri come risolvere il problema dell'inviluppo convesso in tempo reale con un tempo complessivo di $O(n^2)$.

* *35.3-6* Si mostri come realizzare il metodo incrementale per calcolare l'inviluppo convesso di n punti in modo che impieghi tempo $O(n \lg n)$

35.4 Ricerca della coppia di punti più vicini

Si consideri ora il problema di trovare la coppia di punti più vicini in un insieme Q di $n \geq 2$ punti. La "vicinanza" si riferisce alla distanza euclidea: la distanza tra i punti $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ è $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Due punti nell'insieme Q possono essere coincidenti, nel qual caso la distanza tra essi è nulla. Questo problema trova, per esempio, applicazioni nel qual caso la distanza tra essi è nulla. Questo problema trova, per esempio, applicazioni nel sistema di controllo del traffico. Un sistema per il controllo del traffico aereo o marittimo potrebbe aver bisogno di conoscere quali sono i due mezzi più vicini per individuare potenziali collisioni.

Un algoritmo "brutale" per trovare la coppia di punti più vicini controlla semplicemente tutte le $\binom{n}{2} = \Theta(n^2)$ coppie di punti. In questo paragrafo si descriverà per questo problema un algoritmo divide-et-impera il cui tempo di esecuzione è descritto dalla nota ricorrenza $T(n) = 2T(n/2) + O(n)$. Perciò questo algoritmo usa soltanto tempo $O(n \lg n)$.

Algoritmo divide-et-impera

Ogni chiamata ricorsiva dell'algoritmo richiede come input un sottoinsieme $P \subseteq Q$ e gli array X e Y, ognuno dei quali contiene tutti i punti del sottoinsieme P di input. I punti nell'array X sono ordinati in modo tale che le loro ascisse siano monotone crescenti. Analogamente, l'array Y è ordinato rispetto alle ordinate in modo monotono crescente. Si noti che per rientrare nel limite di tempo $O(n \lg n)$, non ci si può permettere di ordinare X e Y ad ogni chiamata ricorsiva: se lo facessimo la ricorrenza del tempo di esecuzione sarebbe $T(n) = 2T(n/2) + O(n \lg n)$, la cui soluzione è $T(n) = O(n \lg^2 n)$. Si vedrà fra poco come usare un "preordinamento" per mantenere questa proprietà senza, di fatto, ordinare ad ogni chiamata ricorsiva.

Una data chiamata ricorsiva con input P, X e Y controlla prima se $|P| \leq 3$. Se è così, la chiamata esegue semplicemente il metodo "brutale" descritto precedentemente: si cercano tutte le $\binom{|P|}{2}$ coppie di punti e si restituisce la coppia dei più vicini. Se $|P| > 3$, la chiamata ricorsiva applica il paradigma divide-et-impera come segue.

Divide: Si trova una linea verticale l che suddivida l'insieme P di punti in due insiemi P_L e P_R tali che $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, tutti i punti in P_L sono a sinistra o sopra la linea l e tutti i punti di P_R sono a destra o sopra la linea l. L'array X è diviso negli array X_L ed X_R che contengono rispettivamente i punti di P_L e P_R, con le ascisse in ordine monotono crescente. Analogamente l'array Y è diviso negli array Y_L ed Y_R che contengono rispettivamente i punti di P_L e P_R, con le ordinate disposte in modo monotono crescente.

Impera: Avendo diviso P in P_L e P_R, si eseguono due chiamate ricorsive, una per trovare la coppia dei punti più vicini in P_L e l'altra per trovare la coppia dei punti più vicini in P_R. Gli input della prima chiamata sono il sottoinsieme P_L e gli array X_L ed Y_L; la seconda chiamata riceve gli input P_R, X_R ed Y_R. Siano δ_L e δ_R le distanze delle coppie dei più vicini restituite per P_L e P_R rispettivamente e sia $\delta = \min(\delta_L, \delta_R)$.

Combina: La coppia dei più vicini è o la coppia con distanza δ trovata da una delle chiamate ricorsive, o una coppia di punti con un punto in P_L e l'altro in P_R. L'algoritmo determina se vi è una tale coppia la cui distanza sia minore di δ. Si osservi che se vi è una coppia di punti con distanza inferiore a δ, entrambi i punti della coppia non devono essere distanti dalla linea l per più di δ unità. Perciò, come mostra la figura 35.12(a), entrambi devono trovarsi nella striscia verticale di ampiezza 2δ centrata sulla linea l. Per trovare una tale coppia, se esiste, l'algoritmo fa quanto segue.

1. Crea un array Y' che è l'array Y da cui sono stati rimossi tutti i punti non nella striscia verticale di ampiezza 2δ. L'array Y' è ordinato rispetto alle ordinate, proprio come Y.

Figura 35.12 *Concetti chiave della dimostrazione che l'algoritmo della coppia dei più vicini ha bisogno di controllare solo i 7 punti che seguono ogni punto nell'array Y'.* (a) *Se $p_L \in P_L$ e $p_r \in P_R$ sono distanti per meno di δ unità essi devono stare all'interno di un rettangolo $\delta \times 2\delta$ centrato sulla linea l.* (b) *Come possono stare dentro un quadrato $\delta \times \delta$ 4 punti che sono distanti, a coppie, almeno δ unità. A sinistra ci sono 4 punti in P_L e a destra ci sono 4 punti in P_R. Vi possono essere 8 punti nel rettangolo $\delta \times 2\delta$ se i punti sulla linea l sono in effetti coppie di punti coincidenti con un punto in P_L e uno in P_R.*

2. Per ogni punto p nell'array Y', l'algoritmo cerca di trovare i punti in Y' che siano distanti da p meno di δ unità. Come si vedrà tra poco, devono essere considerati solo i 7 punti che seguono p. L'algoritmo calcola la distanza tra p ed ognuno di questi 7 punti e tiene traccia della distanza δ' della coppia di punti più vicini trovata su tutte le coppie di punti in Y'.

3. Se $\delta' < \delta$, la striscia verticale contiene in effetti una coppia di punti più vicini di quelli trovati con le chiamate ricorsive; allora vengono restituite questa coppia e la distanza δ'. In caso contrario, vengono restituite la coppia dei più vicini e la distanza δ trovata dalle chiamate ricorsive.

La descrizione precedente omette alcuni dettagli di realizzazione che sono necessari a raggiungere il tempo di esecuzione $O(n \lg n)$. Dopo aver dimostrato la correttezza dell'algoritmo, si mostrerà come realizzare l'algoritmo per ottenere il limite di tempo richiesto.

Correttezza

La correttezza di questo algoritmo per la ricerca della coppia dei più vicini è ovvia, eccetto che per due aspetti. Primo, eliminando la ricorsione quando $|P| \leq 3$, si assicura che non si dividerà mai un insieme di punti composto di un solo punto. Il secondo aspetto è che si ha bisogno di controllare per ogni punto soltanto i 7 punti che seguono p nell'array Y'; si dimostrerà ora questa proprietà.

Si supponga che a qualche livello di ricorsione, la coppia dei punti più vicini sia $p_L \in P_L$ e $p_r \in P_R$. Quindi, la distanza δ' tra p_L e p_R è strettamente minore di δ. Il punto p_L deve essere a sinistra o sopra la linea l e distante da essa per meno di δ unità. Analogamente, p_R deve essere a destra o sopra la linea l e distante da essa per meno di δ unità. Inoltre p_L e p_R sono distanti per meno di δ anche verticalmente. Pertanto, come mostra la figura 35.12(a), p_L e p_R sono

all'interno di un rettangolo $\delta \times 2\delta$ centrato sulla linea l. (Dentro questo rettangolo vi possono essere altri punti.)

Si mostra quindi che al più 8 punti di P possono stare dentro questo rettangolo $\delta \times 2\delta$. Si consideri il quadrato $\delta \times \delta$ che forma la metà sinistra di questo rettangolo. Poiché tutti i punti dentro P_L sono distanti almeno δ unità, al più quattro punti possono risiedere dentro questo quadrato; la figura 35.12(b) mostra come. Analogamente al più 4 punti in P_R possono stare dentro il quadrato $\delta \times \delta$ che forma la metà destra del rettangolo. Per cui al più 8 punti di P possono stare dentro il rettangolo $\delta \times 2\delta$. (Si noti che poiché i punti sulla linea l possono essere o in P_L o in P_R allora vi possono essere al più 4 punti su l. Questo limite è raggiunto se vi sono due coppie di punti coincidenti, ogni coppia composta da un punto di P_L e un punto di P_R, una coppia è sull'intersezione tra l e il lato in alto del rettangolo, e l'altra coppia si trova dove l interseca il lato in basso del rettangolo.)

Avendo mostrato che al più 8 punti di P possono giacere dentro il rettangolo, è facile vedere che si ha bisogno solo di controllare i 7 punti che seguono ogni punto nell'array Y'. Continuando con l'ipotesi che la coppia dei più vicini sia p_L e p_R, si assuma senza perdita di generalità che p_L preceda p_R nell'array Y'. Quindi, anche se p_L appare il prima possibile in Y' e p_R appare il più tardi possibile, p_R è in una delle 7 posizioni che seguono p_L. Per cui, si è mostrata la correttezza dell'algoritmo per la ricerca della coppia di punti più vicini.

Realizzazione e tempo di esecuzione

Come si è già ricordato, l'obiettivo è di avere come ricorrenza del tempo di esecuzione $T(n) = 2T(n/2) + O(n)$, dove $T(n)$ è, naturalmente, il tempo di esecuzione per un insieme di n punti. La principale difficoltà è quella di assicurare che gli array X_L, X_R, Y_L e Y_R, che sono passati alle chiamate ricorsive, siano ordinati rispetto alla coordinata appropriata e anche che l'array Y' sia ordinato rispetto alla coordinata y. (Si noti che se l'array X ricevuto da una chiamata ricorsiva è già ordinato allora la divisione dell'insieme P in P_L e P_R è ottenuta in tempo lineare.)

Si osservi che, in ogni chiamata, si desidera formare un sottoinsieme ordinato di un array ordinato. Per esempio, una chiamata particolare è quella in cui sono dati l'insieme P e l'array Y, ordinato rispetto alla coordinata y. Avendo partizionato P in P_L e P_R, bisogna determinare gli array Y_L e Y_R che sono ordinati rispetto alla coordinata y. Inoltre questi array devono essere determinati in tempo lineare. Il metodo può essere visto come l'opposto della procedura MERGE del merge sort al paragrafo 1.3.1: si suddivide un array ordinato in due array ordinati. La seguente procedura ne dà un'idea.

```
length[Y_L] ← length[Y_R] ← 0
for i ← 1 to length[Y]
    do if Y[i] ∈ P_L
        then  length[Y_L] ← length[Y_L] + 1
              Y[length[Y_L]] ← Y[i]
        else  length[Y_R] ← length[Y_R] + 1
              Y[length[Y_R]] ← Y[i]
```

Si esaminano semplicemente i punti nell'array Y nell'ordine. Se un punto $Y[i]$ è in P_L, allora lo si aggiunge alla fine dell'array Y_L; altrimenti lo si aggiunge alla fine dell'array Y_R.

Procedure analoghe possono essere usate per determinare gli array X_L, X_R e Y'. L'unico problema rimasto in sospeso è come ottenere i punti ordinati nei primi array. Ciò si può realizzare semplicemente *preordinando* gli array, cioè, si ordinano una volta per tutte *prima* della chiamata ricorsiva iniziale. Questi array ordinati sono passati alla prima chiamata ricorsiva e da lì sono ridotti attraverso le chiamate ricorsive quanto è necessario. Il preordinamento richiede un ulteriore tempo di esecuzione $O(n \lg n)$; in tal modo però ogni passo della ricorsione richiede tempo lineare se non si considerano le chiamate ricorsive. Pertanto, se il tempo di esecuzione di ogni passo ricorsivo è $T(n)$ e il tempo di esecuzione dell'intero algoritmo è $T'(n)$, si ha $T'(n) = T(n) + O(n \lg n)$ e

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{se } n > 3 , \\ O(1) & \text{se } n \le 3 . \end{cases}$$

Dunque, $T(n) = O(n \lg n)$ e $T'(n) = O(n \lg n)$.

Esercizi

35.4-1 Il professor Polveroni ha inventato uno schema secondo cui l'algoritmo per la ricerca della coppia di punti più vicini controlla, per ogni punto, solo i 5 punti ad esso successivi nell'array Y'. L'idea è di mettere sempre i punti sulla linea l nell'insieme P_L. Quindi, non vi possono essere coppie di punti sulla linea l coincidenti con un punto in P_L e uno in P_R. Pertanto, al più 6 punti possono stare nel rettangolo $\delta \times 2\delta$. Qual è l'errore dello schema del professore?

35.4-2 Senza aumentare il tempo di esecuzione asintotico dell'algoritmo, si mostri come assicurare che l'insieme di punti passati alla prima chiamata ricorsiva non contenga punti coincidenti. Si dimostri che è allora sufficiente controllare i punti nelle 6 (non 7) posizioni che seguono ogni punto nell'array Y'.

35.4-3 La distanza tra due punti può essere definita in modo diverso da quello euclideo. Nel piano, la ***distanza*** L_m tra i punti p_1 e p_2 è data da $((x_1 - x_2)^m + (y_1 - y_2)^m)^{1/m}$. La distanza euclidea quindi è la distanza L_2. Si modifichi l'algoritmo per la ricerca della coppia dei più vicini in modo che esso usi la distanza L_1, anche nota come ***distanza Manhattan***.

35.4-4 Dati due punti p_1 e p_2 nel piano, la distanza L_∞ tra essi è $\max(|x_1 - x_2|, |y_1 - y_2|)$. Si modifichi l'algoritmo per la ricerca della coppia di punti più vicini in modo che usi la distanza L_∞.

Problemi

35-1 Strati convessi

Dato un insieme Q di punti del piano, lo ***strato convesso*** di Q è definito induttivamente. Il primo strato convesso di Q consiste di quei punti di Q che sono vertici di $CH(Q)$. Per $i > 1$, Q_i consiste dei punti di Q da cui sono stati rimossi tutti i punti negli strati convessi $1, 2, \ldots, i-1$. Quindi, l'i-esimo strato convesso di Q è $CH(Q_i)$ se $Q_i \neq \emptyset$ ed è indefinito altrimenti.

a. Si dia un algoritmo in tempo $O(n^2)$ per trovare gli strati convessi di un insieme di n punti.

b. Si dimostri che su qualunque modello di calcolo che richiede $\Omega(n \lg n)$ per ordinare n numeri reali, è richiesto tempo $\Omega(n \lg n)$ per calcolare gli strati convessi di un insieme di n punti.

35-2 Strati massimali

Sia Q un insieme di n punti nel piano. Si dice che il punto (x, y) ***domina*** il punto (x', y') se $x \ge x'$ e $y \ge y'$. Un punto in Q che non è dominato da alcun altro punto di Q è detto ***massimale***. Si noti che Q può contenere molti punti massimali che possono essere organizzati in ***strati massimali*** come segue. Il primo livello massimale L_1 è l'insieme di punti massimali di Q. Per $i > 1$, l'i-esimo strato massimale L_i è l'insieme di punti massimali di $Q - \bigcup_{j=1}^{i-1} L_j$.

Si supponga che Q abbia k strati massimali non vuoti e sia y_i l'ordinata del punto più a sinistra in L_i, per $i = 1, 2, \ldots, k$. Per ora, si assuma che non esistano due punti in Q che abbiano le stesse coordinate x e y.

a. Si mostri che $y_1 > y_2 > \ldots > y_k$.

Si consideri un punto (x, y) a sinistra di qualunque punto di Q e per cui y è distinto dall'ordinata di qualunque punto di Q. Sia $Q' = Q \cup \{(x, y)\}$.

b. Sia j il minimo indice tale che $y_j < y$ a meno che $y < y_k$, nel qual caso si pone $j = k + 1$. Si mostri che gli strati massimali di Q' hanno le seguenti proprietà.

- Se $j \le k$, allora gli strati massimali di Q' sono gli stessi di Q tranne che L_j include anche (x, y) come suo nuovo punto più a sinistra.
- Se $j = k + 1$, allora i primi k strati massimali di Q' sono gli stessi di Q ma Q' ha in più un $(k + 1)$-esimo strato massimale non vuoto: $L_{k+1} = \{(x, y)\}$.

c. Si descriva un algoritmo in tempo $O(n \lg n)$ per determinare gli strati massimali di un insieme Q di n punti. (*Suggerimento:* si muova una retta di rastrellamento da destra a sinistra.)

d. Quali difficoltà sorgono se a questo punto si permette che i punti in input abbiano le stesse coordinate x e y? Si suggerisca un modo per risolvere tale problema.

35-3 Fantasmi ed Acchiappafantasmi

Un gruppo di n Acchiappafantasmi sta combattendo con n fantasmi. Ogni Acchiappafantasmi è armato con un fucile a protoni che spara un fascio di protoni capace di annientare un fantasma. Un fascio di protoni va in linea retta e termina quando colpisce un fantasma. Gli

Acchiappafantasmi decidono di seguire la seguente strategia. Ognuno di loro "marcherà" un fantasma, formando n coppie Acchiappafantasma-fantasma, quindi contemporaneamente ogni Acchiappafantasmi sparerà un fascio di protoni al fantasma da lui marcato. Come si sa, è *molto* pericoloso incrociare i fasci di protoni, così gli Acchiappafantasmi devono scegliere accoppiamenti in modo da non far incrociare i fasci di protoni.

Si assuma che la posizione di ogni Acchiappafantasmi e di ogni fantasma sia un punto fissato del piano e che non vi siano tre posizioni collineari.

a. Si deduca che esiste una retta che passa attraverso un Acchiappafantasmi ed un fantasma in modo che il numero di Acchiappafantasmi da un lato della linea sia uguale al numero di fantasmi sullo stesso lato. Si descriva come trovare tale linea in tempo $O(n \lg n)$.

b. Si dia un algoritmo in tempo $O(n^2 \lg n)$ per accoppiare gli Acchiappafantasmi con i fantasmi in modo tale che nessun fascio di protoni si incroci con un altro.

35-4 *Distribuzioni inviluppate sparse*

Si consideri il problema di calcolare l'inviluppo convesso di un insieme di punti del piano che siano stati presi secondo una qualche distribuzione casuale conosciuta. Talvolta, l'inviluppo convesso di n punti presi da una tale distribuzione ha dimensione attesa $O(n^{1-\varepsilon})$ per qualche costante $\varepsilon > 0$. Tale distribuzione è chiamata ***inviluppata sparsa***. Esempi di distribuzioni inviluppate sparse sono le seguenti:

- I punti sono presi uniformemente su un cerchio di raggio unitario. L'inviluppo convesso ha dimensione attesa $\Theta(n^{1/3})$.
- I punti sono presi all'interno di un poligono convesso di k lati, per qualche costante k. L'inviluppo convesso ha dimensione attesa $\Theta(\lg n)$.
- I punti sono presi da una distribuzione normale bidimensionale. L'inviluppo convesso ha dimensione attesa $\Theta(\sqrt{\lg n})$.

a. Dati due poligoni convessi rispettivamente con n_1 ed n_2 vertici, si mostri come calcolare l'inviluppo convesso di tutti gli $n_1 + n_2$ punti in tempo $O(n_1 + n_2)$. (I poligoni possono sovrapporsi.)

b. Si mostri che l'inviluppo convesso di un insieme di n punti presi indipendentemente secondo una distribuzione inviluppata sparsa può essere calcolato in tempo atteso $O(n)$. (*Suggerimento*: si trovino ricorsivamente gli inviluppi convessi dei primi $n/2$ punti e quindi degli altri $n/2$ punti e si combinino i risultati.)

Note al capitolo

Questo capitolo ha trattato schematicamente algoritmi e tecniche di geometria computazionale. Libri sulla geometria computazionale includono quelli di Preparata e Shamos [160] e di Edelsbrunner [60].

Sebbene la geometria sia stata studiata sin dall'antichità, lo sviluppo di algoritmi per risolvere problemi geometrici è relativamente recente. Preparata e Shamos segnalano che le

prime nozioni sulla complessità di un problema furono date da E. Lemoine nel 1902. Egli stava studiando costruzioni euclidee – quelle che usano righello e compasso – e individuò un insieme di cinque primitive: puntamento di un braccio del compasso su un dato punto, puntamento di un braccio del compasso su una data linea, disegno di una circonferenza, passaggio del bordo del righello attraverso un dato punto e disegno di una linea. Lemoine si interessò al numero di primitive necessarie per effettuare una data costruzione; chiamò questa quantità la "semplicità" della costruzione.

L'algoritmo del paragrafo 35.2 che determina se dei segmenti si intersecano è dovuto a Shamos e Hoey [176].

La versione originale dell'algoritmo di Graham si trova in Graham [91]. L'algoritmo di impacchettamento è dovuto a Jarvis [112]. Usando un modello di calcolo ad albero di decisione, Yao [205] dimostrò un limite inferiore $\Omega(n \lg n)$ per il tempo di esecuzione di qualunque algoritmo per l'inviluppo convesso. Quando il numero h dei vertici dell'inviluppo convesso è preso in considerazione, l'algoritmo "prune-and-search" di Kirkpatrick e Seidel [120] che richiede tempo $O(n \lg h)$ è asintoticamente ottimo.

L'algoritmo divide-et-impera di tempo $O(n \lg n)$ per trovare la coppia dei punti più vicini è di Shamos ed appare in Preparata e Shamos [160]. Preparata e Shamos mostrano anche che l'algoritmo è asintoticamente ottimo in un modello ad albero di decisione.

Problemi NP-completi

36

Tutti gli algoritmi che sono stati studiati fino ad ora sono ***algoritmi con tempo polinomiale***: su input di dimensione n: il loro tempo di esecuzione nel caso peggiore è $O(n^k)$ per qualche costante k. È naturale chiedersi se *tutti* i problemi possano essere risolti in tempo polinomiale, ma la risposta è no. Per esempio, ci sono problemi, come il famoso "Problema della Fermata" di Turing, che non possono essere risolti da nessun calcolatore, indipendentemente dal tempo disponibile. Ci sono anche problemi che possono essere risolti, ma non in tempo $O(n^k)$ per qualsiasi costante k. Generalmente si pensa ai problemi risolvibili con algoritmi di tempo polinomiale come a problemi trattabili, e ai problemi che richiedono un tempo super polinomiale come a problemi intrattabili.

L'argomento di questo capitolo, tuttavia, riguarda lo studio di un'interessante classe di problemi, chiamati "NP-completi", il cui stato non è conosciuto. Nessun algoritmo con tempo polinomiale è stato ancora trovato per un problema NP-completo, né qualcuno è stato in grado di fornire un limite inferiore con tempo super polinomiale ad uno qualsiasi di essi. Questo problema è spesso indicato sinteticamente come "P ≠ NP" e costituisce uno dei temi di ricerca più difficili in informatica teorica, sin da quando è stato proposto nel 1971.

La maggior parte degli informatici teorici crede che i problemi NP-completi siano intrattabili. La motivazione è che se un qualsiasi singolo problema NP-completo potesse essere risolto in tempo polinomiale, allora *ogni* problema NP-completo avrebbe un algoritmo di tempo polinomiale. Dato il grande numero di problemi NP-completi che sono stati studiati fino ad oggi, senza nessun progresso verso soluzioni con tempo polinomiale, sarebbe stupefacente che potessero essere tutti risolti in tempo polinomiale.

Per diventare un buon progettista di algoritmi è necessario apprendere i rudimenti della teoria della NP-completezza. Se si può stabilire che un problema è NP-completo, allora si ha quasi la certezza della sua intrattabilità. Come progettista, sarà allora meglio dedicarsi allo sviluppo di un algoritmo approssimato (si veda il Capitolo 37), invece di cercare un algoritmo veloce che risolva il problema in modo esatto. Inoltre, molti problemi naturali e interessanti che apparentemente sembrano non più ardui dell'ordinamento, della ricerca su un grafo o del flusso su di una rete, sono in realtà NP-completi. Quindi è importante acquisire familiarità con questa fondamentale classe di problemi.

Questo capitolo studia gli aspetti della NP-completezza che riguardano più direttamente l'analisi degli algoritmi. Nel paragrafo 36.1 si formalizza la nozione di "problema" e si definisce la classe di complessità P dei problemi di decisione che possono essere risolti in tempo polinomiale. Si vedrà anche come queste nozioni siano collegate alla teoria dei linguaggi formali. Il paragrafo 36.2 definisce la classe NP dei problemi di decisione le cui soluzioni possono essere verificate in tempo polinomiale. In tale paragrafo è anche formulato con precisione il problema P ≠ NP.

Il paragrafo 36.3 mostra come le correlazioni tra problemi possano essere studiate attraverso la "riduzione". In esso si definisce la NP-completezza e si schematizza una dimostrazione del fatto che il problema, chiamato "soddisfattibilità di circuiti", sia NP-completo. Avendo trovato un problema NP-completo, nel paragrafo 36.4 si mostra come si possa dimostrare che altri problemi sono NP-completi in modo decisamente più semplice con la metodologia della riduzione. La metodologia è illustrata mostrando che due problemi di soddisfattibilità di formule sono NP-completi. Nel paragrafo 36.5 si esaminano vari problemi che risultano essere NP-completi.

36.1 Tempo polinomiale

Cominciamo lo studio della NP-completezza formalizzando la nozione di problemi risolvibili in tempo polinomiale. Questi problemi sono generalmente considerati trattabili. La ragione di ciò ha motivazioni filosofiche e non matematiche. Si possono considerare tre diversi argomenti a supporto di questa affermazione.

In primo luogo, benché sia ragionevole riferirsi ad un problema che richiede tempo $O(n^{100})$ come intrattabile, ci sono veramente pochi problemi pratici che richiedono tempo polinomiale di grado così alto. I problemi calcolabili in tempo polinomiale che si incontrano in pratica, generalmente richiedono molto meno tempo.

In secondo luogo, per numerosi modelli computazionali, un problema che può essere risolto in tempo polinomiale con un modello può essere risolto ancora in tempo polinomiale con un altro. Per esempio, la classe dei problemi risolvibili in tempo polinomiale con la macchina seriale ad accesso diretto usata nella maggior parte di questo libro è la stessa classe di problemi risolvibili in tempo polinomiale con le macchine astratte di Turing[1]. Essa coincide anche con la classe di problemi risolvibili in tempo polinomiale su calcolatori paralleli, anche quando il numero di processori cresce in modo polinomiale in relazione alla dimensione dell'input.

In terzo luogo, la classe dei problemi risolvibili in tempo polinomiale ha una serie di interessanti proprietà di chiusura, infatti i polinomi sono chiusi rispetto all'addizione, alla moltiplicazione e alla composizione. Per esempio, se l'output di un algoritmo di tempo polinomiale è passato come input ad un altro algoritmo polinomiale, l'algoritmo composto è polinomiale. Se un algoritmo di tempo polinomiale prevede un numero costante di chiamate a procedure con tempo polinomiale, il tempo di esecuzione dell'algoritmo composto è polinomiale.

Problemi astratti

Per comprendere la classe dei problemi risolvibili in tempo polinomiale, si deve prima avere una nozione formale di ciò che è un "problema". Si definisce un ***problema astratto*** Q come una relazione binaria sull'insieme I delle ***istanze*** del problema e l'insieme S delle ***soluzioni*** del problema. Per esempio, si consideri il problema CAMMINO-MINIMO di trovare il cammino più corto tra due vertici dati di un grafo non pesato e non orientato $G = (V, E)$. Un'istanza di CAMMINO-MINIMO è una tripla fatta da un grafo e due vertici. Una soluzione

[1] Si veda Hopcroft e Ullman [104] o Lewis e Papadimitriou [139] per una trattazione completa del modello della macchina di Turing.

è una sequenza di vertici del grafo dove la sequenza vuota rappresenta il fatto che non esiste alcun cammino. Il problema CAMMINO-MINIMO stesso è la relazione che associa ogni istanza di un grafo e due vertici al cammino più corto che collega i due vertici. Poiché il cammino più breve non è necessariamente unico, una data istanza di un problema può avere più di una soluzione.

Questa formulazione di problema astratto è più generale di quella che occorre per gli obiettivi del capitolo. Per semplicità, la teoria dei problemi NP-completi focalizza l'attenzione sui ***problemi di decisione***: quelli che hanno come soluzione una risposta sì o no. In questo caso, si può vedere un problema di decisione astratto come una funzione che fa corrispondere l'insieme delle istanze I all'insieme delle soluzioni $\{0, 1\}$. Per esempio, un problema di decisione CAMMINO, riconducibile al problema del cammino minimo è, "Dati un grafo $G = (V, E)$, due vertici $u, v \in V$ ed un numero intero k non negativo, esiste in G un cammino da u a v la cui lunghezza sia al più k?" Se $i = \langle G, u, v, k \rangle$ è un'istanza di questo problema di cammino minimo, allora CAMMINO(i) = 1 (sì) se il cammino più corto da u a v ha una lunghezza al più k, CAMMINO(i) = 0 (no) altrimenti.

Molti problemi astratti non sono problemi di decisione, ma piuttosto ***problemi di ottimizzazione***, in cui alcuni valori devono essere massimizzati o minimizzati. Per essere in grado di applicare la teoria della NP-completezza ai problemi di ottimizzazione, questi devono essere riformulati come problemi di decisione. Tipicamente, un problema di ottimizzazione può essere riformulato imponendo una limitazione sul valore da ottimizzare. Per esempio, nella riformulazione del problema del cammino minimo in termini del problema decisionale CAMMINO, si è aggiunta all'istanza del problema la limitazione k.

Benché la teoria dell'NP-completezza imponga la riformulazione dei problemi di ottimizzazione come problemi di decisione, questo requisito non diminuisce l'importanza della teoria. In generale, se un problema di ottimizzazione può essere risolto velocemente, allora si può risolvere velocemente anche il problema di decisione corrispondente. Semplicemente, si confronta il valore ottenuto dalla soluzione del problema di ottimizzazione, con la limitazione fornita come input dal problema di decisione. Perciò se un problema di ottimizzazione è facile allora è altrettanto facile il corrispondente problema di decisione. Riformulando ciò in modo più conveniente per la NP-completezza, si può fornire la prova che un problema di decisione è arduo e che anche il corrispondente problema di ottimizzazione è arduo. Quindi, benché la teoria della NP-completezza si riferisca solo ai problemi di decisione, ha una applicabilità molto più ampia.

Codifica

Se per risolvere un problema si usa un programma, allora le istanze del problema devono essere rappresentate in modo che siano accettabili dal programma. Una ***codifica*** di un insieme S di oggetti astratti è una corrispondenza e da S ad un insieme di stringhe binarie[2]. Per esempio, è nota la codifica dei numeri naturali $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ con le stringhe $\{0, 1, 10, 11, 100, \ldots\}$. Usando questa codifica, $e(17) = 10001$. Qualsiasi persona che abbia considerato la rappresentazione dei caratteri della tastiera di un calcolatore, conosce il codice ASCII o quello EBCDIC. Nel codice ASCII, $e(A) = 1000001$. Anche un oggetto composto può essere codificato come una stringa binaria combinando le rappresentazioni delle sue componenti.

[2] Non è necessario che il codominio di e sia formato da stringhe *binarie*; va bene qualsiasi insieme di stringhe su un alfabeto finito con almeno due elementi.

Poligoni, grafi, funzioni, coppie ordinate, programmi – tutti possono essere codificati come stringhe binarie.

Quindi, un algoritmo per un calcolatore che "risolva" un problema di decisione astratto, in realtà prende in input una codifica dell'istanza del problema. Un problema il cui insieme di istanze è codificato in un insieme di stringhe binarie si chiama ***problema concreto***. Si dice che un algoritmo ***risolve*** un problema concreto in tempo $O(T(n))$ se, quando gli viene fornita un'istanza i del problema di lunghezza $n = |i|$, l'algoritmo può produrre la soluzione al più in tempo $O(T(n))$. Perciò un problema concreto è ***risolvibile in tempo polinomiale*** se esiste un algoritmo per risolverlo in tempo $O(n^k)$ per qualche costante k.

Si può adesso definire formalmente la ***classe di complessità P*** come l'insieme dei problemi di decisione concreti che sono risolvibili in tempo polinomiale.

Per far corrispondere problemi astratti a problemi concreti si possono usare le codifiche. Dato un problema di decisione astratto Q che fa corrispondere ad un insieme di istanze I valori su $\{0, 1\}$, una codifica $e : I \rightarrow \{0, 1\}^*$ può essere usata per ottenere un corrispondente problema di decisione concreto che sarà denotato con $e(Q)$. Se la soluzione di un'istanza di un problema astratto $i \in I$ è $Q(i) \in \{0, 1\}$, allora anche la soluzione della corrispondente istanza del problema concreto $e(i) \in \{0, 1\}^*$ è $Q(i)$. Ci può essere la piccola difficoltà tecnica che alcune stringhe binarie non rappresentano istanze del problema astratto. Per semplicità, si può supporre di far corrispondere a tutte queste stringhe il valore 0. Quindi, il problema concreto produce le stesse soluzioni del problema astratto su istanze di stringhe binarie che rappresentino la codifica di istanze del problema astratto.

Il nostro obiettivo ora è quello di estendere la definizione di problema risolvibile in tempo polinomiale dal problema concreto al problema astratto usando la codifica, ma vorremmo anche che la definizione fosse indipendente dalla particolare codifica. Cioè, l'efficienza nella risoluzione di un problema non dovrebbe dipendere da come il problema è codificato. Sfortunatamente, essa dipende piuttosto strettamente dalla codifica. Per esempio, si supponga che un intero k sia fornito come unico input di un algoritmo, e si supponga che il tempo di esecuzione dell'algoritmo sia $\Theta(k)$. Se l'intero k è fornito in ***unario*** – una stringa di k 1 – allora il tempo di esecuzione dell'algoritmo è $O(n)$ nella dimensione n dell'input, che è un tempo polinomiale. Tuttavia, se si usa la comune rappresentazione binaria dell'intero k, allora la lunghezza dell'input è $n = \lfloor \lg k \rfloor + 1$. In questo caso il tempo di esecuzione dell'algoritmo è $\Theta(k) = \Theta(2^n)$, che è esponenziale rispetto alla dimensione dell'input. Quindi, a seconda della codifica, l'algoritmo è eseguito o in tempo polinomiale o in tempo super polinomiale.

La codifica di un problema astratto è perciò abbastanza importante nella comprensione del tempo polinomiale. In realtà, non si può parlare di risolvibilità di un problema astratto senza prima avere specificato una codifica. Ciononostante, in pratica, se si escludono codifiche "costose" come quella unaria, la codifica di un problema ha scarsa influenza sul fatto che il problema possa essere risolto in tempo polinomiale, oppure no. Per esempio, la rappresentazione in base 3 piuttosto che in base 2 non ha alcun effetto, poiché un intero rappresentato in base 3 può essere convertito in base 2 in tempo polinomiale.

Si dice che una funzione $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ è ***calcolabile in tempo polinomiale*** se esiste un algoritmo A con tempo polinomiale che, dato come input un qualsiasi $x \in \{0, 1\}^*$ produce come output $f(x)$. Per un dato insieme I di istanze di problema, si dice che due codifiche e_1 ed e_2 sono ***correlate polinomialmente*** se esistono due funzioni calcolabili in tempo polinomiale f_{12} e f_{21} tali che per ogni $i \in I$, si abbia $f_{12}(e_1(i)) = e_2(i)$ e $f_{21}(e_2(i)) = e_1(i)$. Cioè, la codifica $e_2(i)$ può essere calcolata dalla codifica $e_1(i)$ attraverso un algoritmo di tempo polinomiale e viceversa. Come è mostrato nel seguente lemma, se due codifiche e_1 ed e_2 di un problema astratto sono correlate polinomialmente, la scelta di quale usare non influenza il fatto che il problema sia o no risolvibile in tempo polinomiale.

Lemma 36.1

Sia Q un problema di decisione astratto su un insieme I di istanze, e siano e_1 e e_2 due codifiche di I correlate polinomialmente. Allora $e_1(Q) \in$ P se e solo se $e_2(Q) \in$ P.

Dimostrazione. È necessario dimostrare solo un verso dell'implicazione dato che il verso contrario è simmetrico. Si supponga, perciò, che $e_1(Q)$ possa essere risolto in tempo $O(n^k)$ per qualche costante k. Inoltre, si supponga che per qualsiasi istanza i del problema, la codifica $e_1(i)$ possa essere calcolata dalla codifica $e_2(i)$ in tempo $O(n^c)$ per qualche costante c, dove $n = |e_2(i)|$. Per risolvere $e_2(Q)$, avendo in ingresso $e_2(i)$, si deve prima calcolare $e_1(i)$ e quindi eseguire l'algoritmo per $e_1(Q)$ su $e_1(i)$. Quanto tempo si impiega a fare tutto questo? La conversione delle codifiche impiega un tempo $O(n^c)$ e perciò $|e_1(i)| = O(n^c)$, poiché l'output di un calcolatore seriale non può essere più lungo del suo tempo di esecuzione: produrre ciascun bit in uscita richiede almeno tempo unitario. Per risolvere il problema con $e_1(i)$ si impiega tempo $O(|e_1(i)|^k) = O(n^{ck})$, che è polinomiale poiché sia c che k sono costanti. ■

Quindi, se un problema astratto ha le istanze codificate in binario o in base 3, ciò non influenza la sua "complessità", cioè il fatto che esso sia risolvibile in tempo polinomiale oppure no; ma se le istanze sono codificate in unario allora la sua complessità può cambiare. Per riuscire a ragionare in modo indipendente dalla codifica, generalmente si assume che le istanze del problema siano codificate secondo un qualunque modello ragionevole e conciso, a meno che non sia esplicitamente specificato altrimenti. Per essere più precisi, si assumerà che la codifica di un intero sia correlata polinomialmente alla sua rappresentazione binaria, e che la codifica di un insieme finito sia correlata polinomialmente alla sua codifica come lista di elementi, chiusi tra parentesi e separati da virgole. (ASCII è un esempio di un tale schema di codifica.) Utilizzando questa codifica "tipo", si possono derivare ragionevoli codifiche per altri oggetti matematici come n-uple, grafi e formule. Per denotare la codifica di un oggetto si racchiuderà l'oggetto tra parentesi angolate; per esempio, $\langle G \rangle$ rappresenta la codifica del grafo G.

Fino a che sarà usata una codifica che sia correlata polinomialmente a questa codifica tipo, si potrà ragionare direttamente su problemi astratti senza fare riferimento ad una particolare codifica, sapendo che la scelta della codifica non influenza il fatto che il problema sia risolvibile in tempo polinomiale. D'ora in poi, si assumerà che tutte le istanze del problema siano codificate con stringhe binarie usando la codifica tipo, a meno che non sia esplicitamente specificato diversamente. Inoltre sarà trascurata la distinzione tra problema astratto e concreto. Tuttavia, si dovrebbe riflettere sui problemi che possono nascere in pratica quando la codifica non è ovvia e può implicare una differenza nella complessità.

Il paradigma dei linguaggi formali

Uno dei vantaggi che si ottengono limitandosi allo studio dei problemi di decisione è il fatto che si può utilizzare facilmente la teoria dei linguaggi formali. Vale la pena, a questo punto, richiamare alcune definizioni di questa teoria. Un ***alfabeto*** Σ è un insieme finito di simboli. Un ***linguaggio*** L su Σ è qualsiasi insieme di stringhe costituite da simboli di Σ. Per esempio, se $\Sigma = \{0, 1\}$, l'insieme $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ è il linguaggio delle

rappresentazioni binarie dei numeri primi. Si denota la ***stringa vuota*** con ε, ed il ***linguaggio vuoto*** con ϕ. Il linguaggio di tutte le stringhe su Σ è denotato con Σ^*. Per esempio, se $\Sigma = \{0, 1\}$, allora $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, ...\}$ è l'insieme di tutte le stringhe binarie. Ogni linguaggio L su Σ è un sottoinsieme di Σ^*.

È prevista tutta una serie di operazioni sui linguaggi. Le operazioni della teoria degli insiemi, come ***unione*** ed ***intersezione***, derivano direttamente dalle definizioni insiemistiche. Si definisce il ***complemento*** di L come $\overline{L} = \Sigma^* - L$. La ***concatenazione*** di due linguaggi L_1 e L_2 è il linguaggio

$$L = \{x_1x_2 : x_1 \in L_1 \text{ e } x_2 \in L_2\}.$$

La ***chiusura*** o ***stella di Kleene*** di un linguaggio L è il linguaggio

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots,$$

dove L^k è il linguaggio ottenuto concatenando L con se stesso k volte.

Dal punto di vista della teoria dei linguaggi, l'insieme delle istanze di qualunque problema di decisione Q è semplicemente l'insieme Σ^*, dove $\Sigma = \{0, 1\}$. Poiché Q è caratterizzato da quelle istanze del problema che producono la risposta 1 (sì), si può considerare Q come un linguaggio L su $\Sigma = \{0, 1\}$, dove

$$L = \{x \in \Sigma^* : Q(x) = 1\}.$$

Per esempio, il problema di decisione CAMMINO ha il corrispondente linguaggio

CAMMINO = $\{(G, u, v, k)$: $G(V, E)$ è un grafo non orientato,
$u, v \in V$,
$k \geq 0$ è un intero, e
esiste un cammino da u a v in G la cui lunghezza è al più $k\}$.

(Quando è utile, spesso si userà lo stesso nome – CAMMINO in questo caso – per indicare sia il problema di decisione che il corrispondente linguaggio.)

Il paradigma dei linguaggi formali ci permette di esprimere in modo conciso la relazione tra problemi di decisione e algoritmi che li risolvono. Si dice che un algoritmo A ***accetta*** o ***riconosce*** una stringa $x \in \{0, 1\}^*$ se, dato x come input, $A(x) = 1$. Il linguaggio ***accettato*** o ***riconosciuto*** da un algoritmo A è l'insieme $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, cioè, l'insieme delle stringhe che l'algoritmo accetta. Un algoritmo A ***rifiuta*** una stringa x se $A(x) = 0$.

Anche se un linguaggio L è riconosciuto da un algoritmo A, l'algoritmo non necessariamente rifiuta una stringa di input $x \notin L$. Per esempio, l'algoritmo può restare in un ciclo infinito. Un linguaggio L è ***deciso*** da un algoritmo A se ogni stringa binaria in L è accettata da A e ogni stringa binaria non in L è rifiutata dall'algoritmo A. Un linguaggio L è ***riconosciuto in tempo polinomiale*** da un algoritmo A se, per qualsiasi stringa x di lunghezza n, l'algoritmo accetta x in tempo $O(n^k)$ per qualche costante k se e solo se $x \in L$. Un linguaggio L è ***deciso in tempo polinomiale*** da un algoritmo A se per qualsiasi stringa $x \in \{0, 1\}^*$ di lunghezza n, l'algoritmo decide, ossia fornisce una risposta corretta sull'appartenenza di x a L in tempo $O(n^k)$ per qualche costante k. Quindi, per riconoscere un linguaggio, un algoritmo necessita soltanto di non sbagliare sulle stringhe di L; ma per decidere il linguaggio, l'algoritmo deve accettare o rifiutare correttamente tutte le stringhe di $\{0, 1\}^*$.

Per esempio, il linguaggio CAMMINO può essere riconosciuto in tempo polinomiale. Un algoritmo di tempo polinomiale, che riconosce questo linguaggio, calcola il cammino più corto tra i vertici u e v in G, usando una ricerca in ampiezza e confrontando poi la distanza ottenuta con k. Se la distanza è al più k, allora l'algoritmo restituisce 1 e si ferma. Altrimenti, l'algoritmo continua la sua esecuzione all'infinito. Il linguaggio CAMMINO non è deciso da questo algoritmo, dato che l'algoritmo non restituisce esplicitamente 0 quando la lunghezza del cammino minimo è più grande di k. Un algoritmo di decisione per CAMMINO deve rifiutare esplicitamente tutte le stringhe che non appartengono a CAMMINO. Per un problema come CAMMINO, è semplice progettare un tale algoritmo di decisione. Per altri problemi, come il problema della Fermata di Turing, esiste un algoritmo di riconoscimento ma non esiste un algoritmo di decisione.

Si può definire informalmente una ***classe di complessità*** come un insieme di linguaggi, per i quali l'appartenenza alla classe è determinata in base ad una ***misura di complessità*** dell'algoritmo che determina se una data stringa x appartiene al linguaggio L, come ad esempio il tempo di esecuzione. In realtà la definizione di classe di complessità è qualcosa di più complicato: il lettore interessato può fare riferimento alla pubblicazione fondamentale di Hartmanis e Stearns [95].

Usando il paradigma della teoria dei linguaggi, si può fornire una definizione alternativa della classe di complessità P:

$$\mathrm{P} = \{L \subseteq \{0, 1\}^* : \text{esiste un algoritmo } A \text{ che decide } L \text{ in tempo polinomiale}\}.$$

In effetti P è anche la classe dei linguaggi che possono essere *riconosciuti* in tempo polinomiale.

Teorema 36.2

$\mathrm{P} = \{L : L$ è riconosciuto da un algoritmo di tempo polinomiale$\}$.

Dimostrazione. Poiché la classe dei linguaggi decisi da algoritmi di tempo polinomiale è un sottoinsieme della classe dei linguaggi riconosciuti da algoritmi di tempo polinomiale, è necessario dimostrare solo che se L è riconosciuto da un algoritmo di tempo polinomiale, esso è decidibile da un algoritmo di tempo polinomiale. Sia L il linguaggio riconosciuto da un qualche algoritmo A con tempo polinomiale. Si userà un classico argomento di "simulazione" per definire un algoritmo A' per il quale L risulta anche decidibile. Poiché A riconosce L in tempo $O(n^k)$ per qualche costante k, esiste anche una costante c tale che A riconosce L in al più $T = cn^k$ passi. Per qualsiasi stringa di input x, l'algoritmo A' simula le azioni di A per il tempo T. Alla fine del tempo T, l'algoritmo A' verifica il comportamento di A. Se A ha accettato x, allora A' accetta x restituendo un 1. Se A non ha accettato x, allora A' rifiuta x restituendo 0. Il tempo impiegato da A' per simulare A non aumenta il tempo di esecuzione per più di un fattore polinomiale e quindi il linguaggio L è deciso dall'algoritmo A' in tempo polinomiale. ■

Si noti che la dimostrazione del Teorema 36.2 non è costruttiva. Per un dato linguaggio $L \in P$, in realtà non è detto che sia nota una limitazione del tempo di esecuzione dell'algoritmo A che riconosce L. Malgrado ciò, sappiamo che una tale limitazione deve esistere e perciò esiste un algoritmo A' che può verificare questo limite, anche se possiamo non essere in grado di trovare l'algoritmo A'.

Esercizi

36.1-1 Si definisca il problema di ottimizzazione LUNGHEZZA-DEL-CAMMINO-MASSIMO come la relazione che associa ciascuna istanza di un grafo non orientato e due vertici alla lunghezza del cammino semplice più lungo tra i due vertici. Si definisca il problema di decisione CAMMINO-MASSIMO = $\{\langle G, u, v, k\rangle : G(V, E)$ è un grafo non orientato, $u, v \in V$, $k \geq 0$ è un intero, ed esiste un cammino semplice da u a v in G la cui lunghezza è almeno $k\}$. Si mostri che il problema di ottimizzazione LUNGHEZZA-DEL-CAMMINO-MASSIMO può essere risolto in tempo polinomiale se e solo se CAMMINO-MASSIMO $\in P$.

36.1-2 Si fornisca una definizione formale del problema di trovare il ciclo semplice più lungo in un grafo non orientato. Si determini il corrispondente problema di decisione. Si definisca il linguaggio corrispondente al problema di decisione.

36.1-3 Si determini una codifica formale di un grafo orientato in termini di stringhe binarie usando una rappresentazione con matrice di adiacenza. Si faccia lo stesso usando una rappresentazione con lista di adiacenza. Si mostri che le due rappresentazioni sono correlate polinomialmente.

36.1-4 L'algoritmo di programmazione dinamica per il problema dello zaino 0-1, richiesto nell'Esercizio 17.2-2, è un algoritmo di tempo polinomiale? Si giustifichi la risposta.

36.1-5 Si supponga che per un certo linguaggio L vi sia un algoritmo che accetta qualsiasi stringa $x \in L$ in tempo polinomiale, ma l'algoritmo sia eseguito in tempo super polinomiale quando $x \notin L$. Si mostri che L può essere deciso da un algoritmo di tempo polinomiale.

36.1-6 Si mostri che un algoritmo che effettua operazioni in tempo polinomiale e al più un numero costante di chiamate a procedure di tempo polinomiale, è eseguito in tempo polinomiale, ma che un numero polinomiale di chiamate a procedure polinomiali può comportare un algoritmo di tempo esponenziale.

36.1-7 Si mostri che la classe P, vista come un insieme di linguaggi, è chiusa rispetto a unione, intersezione, concatenazione, complemento e stella di Kleene. Cioè se L_1 e $L_2 \in P$, allora $L_1 \cup L_2 \in P$, ecc.

36.2 Verifica in tempo polinomiale

Si considerano adesso algoritmi che "verificano" l'appartenenza a un linguaggio. Per esempio, si supponga che per una data istanza $\langle G, u, v, k\rangle$ del problema di decisione CAMMINO sia fornito anche un cammino p da u a v. Si può facilmente verificare se la lunghezza di p è al più k, e se è così, allora si può considerare p come un "certificato" che l'istanza indubbiamente appartiene a CAMMINO. Per il problema di decisione CAMMINO,

(a) (b)

Figura 36.1 (a) *Un grafo che rappresenta i vertici, gli archi e le facce di un dodecaedro, con un ciclo hamiltoniano mostrato con gli archi in grigio.* (b) *Un grafo bipartito con un numero dispari di vertici. Qualsiasi grafo di questo tipo non è hamiltoniano.*

questo certificato non sembra servirci molto: dopo tutto, CAMMINO appartiene a P – in effetti CAMMINO può essere risolto in tempo lineare – e quindi la verifica dell'appartenenza attraverso un dato certificato impiega lo stesso tempo della risoluzione del problema partendo da zero. Si esaminerà adesso un problema per il quale non risulta noto alcun algoritmo di decisione con tempo polinomiale, eppure, dato un certificato, il problema della verifica è semplice.

Cicli hamiltoniani

Il problema di trovare un ciclo hamiltoniano in un grafo non orientato è studiato da oltre un secolo. Formalmente, un ***ciclo hamiltoniano*** di un grafo non orientato $G(V,E)$ è un ciclo semplice che contiene ciascun vertice di V. Un grafo che contiene un ciclo hamiltoniano è detto ***hamiltoniano***; altrimenti esso è ***non hamiltoniano***. Bondy e Murty [31] citano una lettera di W. R. Hamilton che descrive un gioco matematico su un dodecaedro (figura 36.1(a)) in cui un giocatore mette cinque birilli in altrettanti vertici consecutivi scelti a piacere, mentre l'altro giocatore deve completare il cammino per formare un ciclo che contenga tutti i vertici. Il dodecaedro è hamiltoniano, e la figura 36.1(a) mostra un ciclo hamiltoniano. Tuttavia, non tutti i grafi sono hamiltoniani. Per esempio, la figura 36.1(b) mostra un grafo bipartito con un numero dispari di vertici. (L'Esercizio 36.2-2 chiede di mostrare che tutti i grafi di questo tipo non sono hamiltoniani.)

Si può definire il ***problema del ciclo hamiltoniano***, "Un grafo G ha un ciclo hamiltoniano?", con il linguaggio formale:

CICLO-HAMILTONIANO = $\{\langle G\rangle : G$ è un grafo hamiltoniano$\}$.

Quale può essere un algoritmo di decisione per il linguaggio CICLO-HAMILTONIANO?

Data un'istanza $\langle G \rangle$ del problema, un possibile algoritmo di decisione dovrebbe elencare tutte le permutazioni dei vertici di G e poi verificare ciascuna di esse per vedere se è un cammino hamiltoniano. Qual è il tempo di esecuzione di questo algoritmo? Se si usa la codifica "ragionevole" di un grafo attraverso la sua matrice di adiacenza, il numero m dei vertici del grafo è $\Omega(\sqrt{n})$, dove $n = |\langle G \rangle|$ è la lunghezza della codifica di G. Ci sono $m!$ possibili permutazioni dei vertici e perciò il tempo di esecuzione è $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$, che non è $O(n^k)$ per nessuna costante k. Quindi, questo algoritmo ingenuo non è eseguito in tempo polinomiale, ed in effetti, il problema del ciclo hamiltoniano è NP-completo, come si dimostrerà nel paragrafo 36.5.

Algoritmi di verifica

Si consideri tuttavia un problema leggermente più semplice. Si supponga che un amico sostenga che un dato grafo G è hamiltoniano e che, per dimostrarlo, fornisca i vertici ordinati che fanno parte del ciclo hamiltoniano. È certamente semplice verificare la sua dimostrazione: semplicemente si controlla che il ciclo fornito sia hamiltoniano, verificando che è una permutazione dei vertici di V e che ogni arco consecutivo del ciclo esiste veramente nel grafo. Questo algoritmo di verifica può essere certamente realizzato ed eseguito in tempo $O(n^2)$, dove n è la lunghezza della codifica di G. Quindi, la dimostrazione che in un grafo esiste un ciclo hamiltoniano può essere verificata in tempo polinomiale.

Si definisce ***algoritmo di verifica*** un algoritmo A a due argomenti, dove un argomento è una comune stringa x di input e l'altro è una particolare stringa binaria y chiamata ***certificato***. Si dice che un algoritmo A a due argomenti ***verifica*** una stringa x di input se esiste un certificato y tale che $A(x,y) = 1$. Il ***linguaggio verificato*** da un algoritmo A di verifica è

$$L = \{x \in \{0, 1\}^* : \text{esiste } y \in \{0, 1\}^* \text{ tale che } A(x, y) = 1\}.$$

Intuitivamente, un algoritmo A verifica un linguaggio L se per ogni stringa $x \in L$, c'è un certificato y che A può usare per dimostrare che $x \in L$. Inoltre, per qualunque stringa $x \notin L$, non ci devono essere certificati che dimostrino che $x \in L$. Per esempio, nel problema del ciclo hamiltoniano, il certificato è la lista dei vertici nel ciclo hamiltoniano. Se un grafo è hamiltoniano, il ciclo hamiltoniano stesso offre informazioni sufficienti a verificarlo. Viceversa, se un grafo non è hamiltoniano, non esiste nessuna lista di vertici che possa far credere ad un algoritmo di verifica che il grafo è hamiltoniano, dato che l'algoritmo di verifica per essere sicuro, controlla sistematicamente il ciclo proposto.

La classe di complessità NP

La ***classe di complessità*** **NP** è la classe dei linguaggi che possono essere verificati mediante un algoritmo di tempo polinomiale[3]. Più precisamente, un linguaggio L appartiene ad NP se e solo se esistono un algoritmo A a due argomenti di tempo polinomiale ed una costante c tale che

$$L = \{x \in \{0, 1\}^* : \text{esiste un certificato } y \text{ con } |y| = O(|x|^c), \text{ tale che } A(x, y) = 1\}.$$

Si dice che l'algoritmo A ***verifica*** il linguaggio L ***in tempo polinomiale***.

[3] Il nome "NP" sta per "tempo polinomiale non deterministico". La classe NP fu originariamente studiata nel contesto del non determinismo, ma in questo testo si usa una nozione più semplice, quella equivalente di verifica. Hopcroft e Ullman [104] forniscono una buona presentazione della NP-completezza in termini di modelli di calcolo non deterministici.

Figura 36.2 *Le quattro possibili relazioni tra le classi di complessità. In ciascun diagramma, una regione che ne racchiude un'altra indica una relazione di inclusione propria tra insiemi.* (a) P = NP = co-NP. *La maggior parte dei ricercatori considera questa possibilità molto improbabile.* (b) *Se* NP *è chiuso rispetto all'operazione di complemento, allora* NP = co-NP, *ma non necessariamente* P = NP. (c) P = NP ∩ co-NP *ma* NP *non è chiuso rispetto all'operazione di complemento.* (d) NP ≠ co-NP *e* P ≠ NP ∩ co-NP. *La maggior parte dei ricercatori indica questa come la possibilità più probabile.*

Dalla precedente analisi del problema del ciclo hamiltoniano, segue che CICLO-HAMILTONIANO ∈ NP. (È sempre confortante sapere che un insieme importante non è vuoto.) Inoltre, se $L \in$ P allora $L \in$ NP, poiché, se esiste un algoritmo di tempo polinomiale che decide il linguaggio L, allora l'algoritmo può essere facilmente trasformato in un algoritmo di verifica a due input che semplicemente ignora qualsiasi certificato ed accetta esattamente quelle stringhe che esso determina appartenere ad L. Quindi P ⊆ NP.

Non si conosce se P = NP, tuttavia molti ricercatori credono che P ed NP non siano la stessa classe. Intuitivamente, la classe P è costituita dai problemi che possono essere risolti velocemente. La classe NP è costituita dai problemi per i quali una soluzione può essere facilmente verificata. Spesso, in base all'esperienza pratica, sembra che risolvere un problema dall'inizio risulti più difficile che verificare una soluzione presentata esplicitamente. Gli informatici teorici, generalmente, ritengono che questa analogia si possa estendere alle classi P ed NP e che quindi NP includa linguaggi che non sono in P.

C'è una evidenza ancora più convincente che P ≠ NP: l'esistenza di linguaggi "NP-completi". Si studierà questa classe nel paragrafo 36.3.

Molte altre domande fondamentali, oltre al quesito P ≠ NP, rimangono aperte. Malgrado il gran lavoro di molti ricercatori, nessuno sa se la classe NP è chiusa rispetto all'operazione di complemento. Cioè, $L \in$ NP implica $\overline{L} \in$ NP? Si può definire la ***classe di complessità* co-NP** come l'insieme dei linguaggi L tali che $\overline{L} \in$ NP. Il quesito se NP è chiuso rispetto all'operazione di complemento, può essere riformulato nella domanda "NP = co-NP?". Poiché P è chiuso rispetto all'operazione di complemento, (Esercizio 36.1-7), segue che P ⊆ NP ∩ co-NP. Tuttavia, di nuovo, non è noto se P = NP ∩ co-NP o se c'è qualche linguaggio in NP ∩ co-NP − P. La figura 36.2 mostra i quattro possibili casi.

Per concludere, la comprensione della relazione precisa tra P e NP è tristemente incompleta. Ciononostante, esaminando la teoria della NP-completezza, si troverà che lo svantaggio nel dimostrare che un problema è intrattabile, da un punto di vista pratico non è così pesante come si potrebbe supporre.

Esercizi

36.2-1 Si consideri il linguaggio ISOMORFISMO-DI-GRAFI = $\{\langle G_1, G_2\rangle : G_1$ e G_2 sono grafi isomorfi$\}$. Si dimostri che ISOMORFISMO-DI-GRAFI $\in$ NP descrivendo un algoritmo di tempo polinomiale che verifichi il linguaggio.

36.2-2 Si dimostri che se G è un grafo non orientato bipartito con un numero dispari di vertici, allora G è non hamiltoniano.

36.2-3 Si mostri che, se CICLO-HAMILTONIANO $\in$ P, allora il problema di elencare in ordine i vertici di un ciclo hamiltoniano è risolvibile in tempo polinomiale.

36.2-4 Si dimostri che la classe dei linguaggi NP è chiusa rispetto alle operazioni di unione, intersezione, concatenazione e stella di Kleene. Si discuta la chiusura di NP rispetto all'operazione di complemento.

36.2-5 Si mostri che qualsiasi linguaggio di NP può essere deciso con un algoritmo che è eseguito in tempo $2^{O(n^k)}$ per qualche costante k.

36.2-6 Un ***cammino hamiltoniano*** in un grafo è un cammino semplice che visita tutti i vertici esattamente una volta. Si mostri che il linguaggio CAMMINO-HAMILTONIANO = $\{\langle G, u, v\rangle$: c'è un cammino hamiltoniano da u a v nel grafo $G\}$ appartiene a NP.

36.2-7 Si mostri che il problema del cammino hamiltoniano può essere risolto in tempo polinomiale per un grafo orientato senza cicli. Si dia un algoritmo efficiente per questo problema.

36.2-8 Sia ϕ una formula booleana costruita con le variabili booleane di input $x_1, x_2, \ldots, x_k$, la negazione ($\neg$), AND ($\wedge$), OR ($\vee$) e parentesi. Questa formula ϕ è una ***tautologia*** se vale 1 per ogni assegnamento di 1 e 0 alle variabili di input. Si definisca TAUTOLOGIA come il linguaggio delle formule booleane che sono tautologie. Si mostri che TAUTOLOGIA $\in$ co-NP.

36.2-9 Si dimostri che P $\subseteq$ co-NP.

36.2-10 Si dimostri che se P $\neq$ co-NP, allora P $\neq$ NP.

36.2-11 Sia G un grafo non orientato connesso con almeno 3 vertici, e sia G^3 il grafo ottenuto connettendo tutte le coppie di vertici che in G sono connessi da un cammino di lunghezza al più 3. Si dimostri che G^3 è hamiltoniano. (*Suggerimento:* si costruisca un albero di copertura per G, e si usi una dimostrazione per induzione.)

36.3 NP-completezza e riducibilità

Forse la ragione più fondata per la quale molti ricercatori credono che sia P≠NP è l'esistenza della classe dei problemi "NP-completi". Questa classe ha la sorprendente proprietà che se *un qualsiasi* problema NP-completo può essere risolto in tempo polinomiale, allora *tutti* i problemi di NP possono essere risolti in tempo polinomiale, cioè P = NP. Malgrado anni di studio, tuttavia, non è stato trovato alcun algoritmo con tempo polinomiale per un qualsiasi problema NP-completo.

Il linguaggio CICLO-HAMILTONIANO è un problema NP-completo. Se CICLO-HAMILTONIANO potesse essere deciso da un algoritmo con tempo polinomiale allora si potrebbero risolvere tutti i problemi di NP in tempo polinomiale. Infatti, se NP – P dovesse risultare non vuoto, si potrebbe dire con certezza che CICLO-HAMILTONIANO $\in$ NP – P.

I linguaggi NP-completi sono, in un certo senso, i linguaggi "più ardui" in NP. In questo paragrafo si mostrerà come confrontare la complessità relativa di linguaggi usando una nozione precisa chiamata "riducibilità in tempo polinomiale". Per prima cosa, si definiscono formalmente i linguaggi NP-completi, e poi si abbozza la dimostrazione che un linguaggio, chiamato SODDISFATTIBILITÀ-DI-CIRCUITI, è NP-completo. Nel paragrafo 36.5 si userà la nozione di riducibilità per mostrare che molti altri problemi sono NP-completi.

Riducibilità

Intuitivamente, un problema Q può essere ridotto ad un altro problema Q' se una qualsiasi istanza di Q può essere "facilmente riformulata" come un'istanza di Q', e la soluzione di quest'ultima fornisce una soluzione all'istanza di Q. Per esempio, il problema di risolvere equazioni lineari in un'incognita x, si riduce al problema di risolvere equazioni quadratiche. Data un'istanza $ax + b = 0$, si trasforma in $0x^2 + ax + b = 0$, la cui soluzione fornisce una soluzione di $ax + b = 0$. Quindi, se un problema Q si riduce ad un altro problema Q', allora Q, in un certo senso, è "non più arduo da risolvere" di Q'.

Tornando al paradigma dei linguaggi formali per i problemi di decisione, si dice che un linguaggio L_1 è ***riducibile in tempo polinomiale*** ad un linguaggio L_2, e si scrive $L_1 \leq_P L_2$, se esiste una funzione calcolabile in tempo polinomiale $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ tale che per ogni $x \in \{0, 1\}^*$,

$$x \in L_1 \text{ se e solo se } f(x) \in L_2. \tag{36.1}$$

La funzione f si chiama ***funzione di riduzione***, ed un algoritmo F di tempo polinomiale che calcola f è detto ***algoritmo di riduzione***.

La figura 36.3 illustra l'idea di una riduzione in tempo polinomiale da un linguaggio L_1 ad un linguaggio L_2. Ciascun linguaggio è un sottoinsieme di $\{0, 1\}^*$. La funzione di riduzione f fornisce in tempo polinomiale un'applicazione per cui, se $x \in L_1$, allora $f(x) \in L_2$. Inoltre, se $x \notin L_1$ allora $f(x) \notin L_2$. Quindi, la funzione di riduzione fa corrispondere qualsiasi istanza x del problema di decisione rappresentato col linguaggio L_1, ad un'istanza $f(x)$ del problema di decisione associato a L_2. Decidere se $f(x) \in L_2$ consente di decidere anche se $x \in L_1$.

Figura 36.3 *Illustrazione di una riduzione in tempo polinomiale da un linguaggio L_1 ad un linguaggio L_2 mediante una funzione di riduzione f. Per qualsiasi input $x \in \{0, 1\}^*$, la domanda se $x \in L_1$, ha la stessa risposta della domanda se $f(x) \in L_2$.*

Le riduzioni in tempo polinomiale rappresentano uno strumento potente per dimostrare che diversi linguaggi appartengono a P.

Lemma 36.3

Se $L_1, L_2 \subseteq \{0,1\}^*$ sono linguaggi tali che $L_1 \leq_P L_2$, allora $L_2 \in$ P implica $L_1 \in$ P.

Dimostrazione. Sia L_2 un linguaggio deciso da un algoritmo A_2, e sia F un algoritmo di riduzione con tempo polinomiale che calcola la funzione di riduzione f. Si vuol costruire un algoritmo di decisione A_1 per il linguaggio L_1.

La figura 36.4 illustra la costruzione di A_1. Per un dato input $x \in \{0, 1\}^*$, l'algoritmo A_1 usa F per trasformare x in $f(x)$, e quindi usa A_2 per verificare se $f(x) \in L_2$. L'output di A_2 è il valore fornito come output da A_1.

La correttezza di A_1 segue dalla condizione (36.1). L'algoritmo è eseguito in tempo polinomiale, poiché sia F che A_2 sono eseguiti in tempo polinomiale (si veda l'Esercizio 36.1-6). ■

NP-completezza

Le riduzioni in tempo polinomiale forniscono uno strumento formale per mostrare che un problema è arduo almeno quanto un altro, a meno di un fattore di tempo polinomiale. Cioè, se $L_1 \leq_P L_2$, allora L_1 non è più arduo di L_2 per più di un fattore polinomiale: ecco perché la notazione "minore o uguale" per la riduzione risulta così mnemonica. Si può adesso definire l'insieme dei linguaggi NP-completi, che sono i problemi più ardui in NP.

Un linguaggio $L \subseteq \{0, 1\}^*$ è ***NP-completo*** se

1. $L \in$ NP, e
2. $L' \leq_P L$ per ogni $L' \in$ NP.

Se un linguaggio L soddisfa la proprietà 2, ma non necessariamente la proprietà 1, si dice che L è ***NP-arduo***. Si definisce anche NPC come la classe dei linguaggi NP-completi.

Come mostra il seguente teorema, la NP-completezza è cruciale per decidere se P è in effetti uguale a NP.

Figura 36.4 *La dimostrazione del lemma 36.3. L'algoritmo F è un algoritmo di riduzione che calcola la funzione di riduzione f da L_1 a L_2 in tempo polinomiale, e A_2 è un algoritmo di decisione con tempo polinomiale per L_2. È illustrato un algoritmo A_1 che stabilisce se $x \in L_1$ usando F per trasformare qualsiasi input x in f(x) e poi usando A_2 per stabilire se $f(x) \in L_2$.*

Teorema 36.4

Se un qualsiasi problema NP-completo è risolvibile in tempo polinomiale, allora P = NP. Equivalentemente, se un qualsiasi problema in NP non è risolvibile in tempo polinomiale, allora tutti i problemi NP-completi non sono risolvibili in tempo polinomiale.

Dimostrazione. Si supponga che $L \in$ P ed anche che $L \in$ NPC. Per qualsiasi $L' \in$ NP, si ha che $L' \leq_P L$ per la proprietà 2 della definizione di NP-completezza. Quindi, per il lemma 36.3, si ha anche che $L' \in$ P, il che dimostra la prima asserzione del teorema.

Per dimostrare la seconda asserzione, che è vera, essendo la contronominale della prima, si supponga che esista un $L \in$ NP tale che $L \notin$ P. Sia $L' \in$ NPC un qualsiasi linguaggio NP-completo e, per assurdo, si ipotizzi che $L' \in$ P. Ma allora, per il lemma 36.3 si ha che $L \leq_P L'$, e quindi $L \in$ P. ■

È questa la ragione per cui la ricerca sul quesito P ≠ NP è centrata sui problemi NP-completi. La maggior parte degli informatici teorici crede che sia P ≠ NP, il che fornisce le relazioni tra P, NP e NPC mostrate nella figura 36.5. Ma, per quanto si sa, qualcuno potrebbe saltar fuori con un algoritmo di tempo polinomiale per un problema NP-completo, dimostrando così che P = NP. Ciononostante, poiché nessun algoritmo di tempo polinomiale è stato ancora trovato per un qualsiasi problema NP-completo, la dimostrazione che un problema è NP-completo fornisce un eccellente indizio della sua intrattabilità.

Soddifattibilità di circuiti

Abbiamo definito la nozione di problema NP-completo, ma finora non abbiamo fornito nessun problema NP-completo. Una volta che si sia dimostrato che almeno un problema è NP-completo, si può usare il teorema della riduzione polinomiale come strumento per dimostrare la NP-completezza di altri problemi. Quindi, si focalizza ora l'attenzione sulla

Figura 36.5 *Come la maggior parte degli informatici teorici vede le relazioni tra* P, NP *e* NPC. *Sia* P *che* NPC *sono completamente contenuti in* NP, *e* P ∩ NPC = ∅.

Figura 36.6 *Due istanze del problema di soddisfattibilità di circuiti.* (a) *L'assegnamento* $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ *per gli input del circuito comporta che l'output del circuito sia* 1. *Il circuito è perciò soddisfattibile.* (b) *Nessun assegnamento per gli input di questo circuito può far in modo che il suo output sia* 1. *Il circuito è perciò non soddisfattibile.*

dimostrazione dell'esistenza di un problema NP-completo: il problema della soddisfattibilità di circuiti.

Sfortunatamente, la dimostrazione formale che il problema della soddisfattibilità di circuiti è NP-completo richiede dettagli tecnici che vanno oltre gli obiettivi di questo testo. Invece, si descriverà informalmente una dimostrazione che si basa sui fondamenti dei circuiti combinatori booleani. Questo materiale è stato esaminato all'inizio del Capitolo 29.

La figura 36.6 mostra due circuiti combinatori booleani, ciascuno con tre input ed un singolo output. Un ***assegnamento di verità*** per un circuito combinatorio booleano è un insieme di valori booleani di input. Si dice che un circuito combinatorio booleano ad un solo output è ***soddisfattibile*** se ha un ***assegnamento di soddisfattibilità***: un assegnamento di verità che comporta il valore 1 come output del circuito. Per esempio, il circuito nella figura 36.6(a) ha assegnamento di soddisfattibilità $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$, e quindi risulta soddisfattibile. Nessun assegnamento di valori a x_1, x_2, x_3 comporta l'output 1 del circuito di figura 36.6(b) che produce sempre 0, e quindi non è soddisfattibile.

Il ***problema della soddisfattibilità di circuiti*** è "Dato un circuito combinatorio booleano composto da porte AND, OR e NOT, esso è soddisfattibile?" Per formulare tale quesito in modo formale, tuttavia, è necessario adeguarsi alla codifica generale dei circuiti. Si può immaginare una codifica basata sui grafi che faccia corrispondere, a un qualsiasi circuito C dato, una stringa binaria $\langle C\rangle$ la cui lunghezza non sia troppo più grande della dimensione del circuito stesso. Rispetto ai linguaggi formali, si può perciò definire

SODDISFATTIBILITÀ-DI-CIRCUITI = $\{\langle C\rangle$: C è un circuito combinatorio booleano soddifattibile$\}$.

Il problema della soddisfattibilità di circuiti ha una rilevante importanza nell'area dell'ottimizzazione dell'hardware con l'aiuto del calcolatore. Se un circuito produce sempre 0, può essere sostituito da un circuito più semplice che elimina tutte le porte logiche e fornisce sempre il valore 0 come output. Un algoritmo di tempo polinomiale per questo problema avrebbe una considerevole applicazione pratica.

Dato un circuito C, si può tentare di determinare se è soddisfattibile semplicemente verificando tutti i possibili assegnamenti per gli input. Sfortunatamente, se ci sono k input, ci sono 2^k possibili assegnamenti. Quando la dimensione di C è polinomiale rispetto a k, la verifica di ciascun input porta ad un algoritmo di tempo super polinomiale. In realtà, come si è già detto, è molto probabile che non esista alcun algoritmo di tempo polinomiale per risolvere il problema della soddisfattibilità, perché la soddisfattibilità di circuiti è un problema NP-completo. Spezzeremo la dimostrazione di ciò in due parti, basandoci sulle due parti della definizione di NP-completezza.

Lemma 36.5

Il problema della soddisfattibilità di circuiti appartiene alla classe NP

Dimostrazione. Si fornirà un algoritmo A a due input con tempo polinomiale che può verificare SODDISFATTIBILITÀ-DI-CIRCUITI. Uno degli input di A è la codifica tipo di un circuito booleano combinatorio C. L'altro input è un certificato corrispondente all'assegnamento di un valore booleano a ciascun filo di C.

L'algoritmo A è costruito come segue. Per ogni porta logica del circuito, verifica che il valore fornito dal certificato sia correttamente calcolato sul filo di uscita come funzione dei valori sui fili di ingresso. Quindi, se l'uscita dell'intero circuito è 1, l'algoritmo restituisce 1, dato che i valori assegnati agli ingressi di C forniscono un assegnamento soddisfattibile. Altrimenti A restituisce 0.

Quando un circuito soddisfattibile C è passato come input all'algoritmo A, esiste un certificato che ha lunghezza polinomiale rispetto alla dimensione di C e che produce in uscita 1. Ogni volta che un circuito non soddisfattibile è dato in input, nessun certificato può far credere ad A che quel circuito sia soddisfattibile. L'algoritmo A è eseguito in tempo polinomiale: con una buona realizzazione, è sufficiente un tempo lineare. Quindi, SODDISFATTIBILITÀ-DI-CIRCUITI può essere verificato in tempo polinomiale, e SODDISFATTIBILITÀ-DI-CIRCUITI $\in$ NP. ■

La seconda parte della dimostrazione che SODDISFATTIBILITÀ-DI-CIRCUITI è NP-completo consiste nel mostrare che il linguaggio è NP-arduo. Cioè, si deve mostrare che ogni linguaggio in NP è riducibile polinomialmente al problema SODDISFATTIBILITÀ-DI-CIRCUITI. La reale dimostrazione di questo fatto è piena di complicazioni tecniche e così preferiamo dare un abbozzo della dimostrazione basato sulla comprensione del funzionamento dell'hardware di un calcolatore.

Un programma per calcolatore è memorizzato nella memoria del calcolatore come una sequenza di istruzioni. Una generica istruzione codifica un'operazione che deve essere elaborata, gli indirizzi degli operandi nella memoria, ed un indirizzo dove va memorizzato

il risultato. Una speciale locazione di memoria, chiamata *contatore del programma*, mantiene traccia della prossima istruzione da eseguire. Il contatore del programma è automaticamente incrementato ogni volta che un'istruzione è caricata, ciò comporta che il calcolatore esegua le istruzioni sequenzialmente. Tuttavia, l'esecuzione di un'istruzione può richiedere che un valore sia scritto nel contatore del programma, e quindi la normale esecuzione sequenziale può essere modificata, permettendo al calcolatore di eseguire cicli ed elaborare rami condizionati.

In qualunque punto dell'esecuzione del programma, l'intero stato del calcolo è rappresentato nella memoria del calcolatore. (Si considera la memoria che contiene il programma stesso, il contatore del programma, la memoria di lavoro e tutti i bit di stato che il calcolatore mantiene per gestire l'esecuzione.) Ogni particolare stato della memoria di un calcolatore si chiama *configurazione*. L'esecuzione di una istruzione può essere considerata come la trasformazione di una configurazione in un'altra. Fondamentalmente, l'hardware del calcolatore che determina questa trasformazione può essere realizzato con circuiti combinatori booleani, che saranno denotati con M nella dimostrazione del seguente lemma.

Lemma 36.6

Il problema della soddisfattibilità di circuiti è NP-arduo.

Dimostrazione. Sia L un qualsiasi linguaggio di NP. Si descriverà un algoritmo F di tempo polinomiale che calcola una funzione di riduzione f che mette in corrispondenza ogni stringa binaria x con un circuito $C = f(x)$ tale che $x \in L$ se e solo se $C \in$ SODDISFATTIBILITÀ-DI-CIRCUITI.

Dato che $L \in$ NP, deve esistere un algoritmo A che verifica L in tempo polinomiale. L'algoritmo F che sarà costruito userà l'algoritmo a due ingressi A per calcolare la funzione di riduzione f.

Sia $T(n)$ il tempo di esecuzione nel caso peggiore dell'algoritmo A su una stringa di input lunga n, e sia $k \geq 1$ una costante tale che $T(n) = O(n^k)$ e la lunghezza del certificato sia $O(n^k)$. (Il tempo di esecuzione di A è in realtà un polinomio rispetto alla dimensione totale dell'input, che include sia la stringa di input che il certificato, ma, dato che il certificato è polinomiale rispetto alla lunghezza n della stringa di input, allora il tempo di esecuzione è polinomiale rispetto ad n.)

L'idea chiave della dimostrazione è quella di rappresentare il calcolo di A come una sequenza di configurazioni. Come mostrato nella figura 36.7, ciascuna configurazione può essere spezzata in più parti che rappresentano: il programma per A, il contatore del programma, altre informazioni di stato ausiliarie, l'input x, il certificato y, e la memoria di lavoro. Partendo da una configurazione iniziale c_0, ciascuna configurazione c_i è trasformata nella configurazione successiva c_{i+1} dal circuito combinatorio M che implementa l'hardware del calcolatore. Il risultato dell'algoritmo A (cioè 0 o 1) è scritto in qualche locazione specifica della memoria di lavoro quando A termina l'esecuzione, e se si assume che dopo A si fermi, allora il valore non cambia. Quindi, se l'algoritmo è eseguito per al più $T(n)$ passi, il risultato è uno dei bit di $c_{T(n)}$.

L'algoritmo di riduzione F costruisce un singolo circuito combinatorio che calcola tutte le configurazioni prodotte a partire da una configurazione iniziale. L'idea è di mettere insieme $T(n)$ copie del circuito M. L'output del circuito i-esimo, che produce la configurazione c_i, è collegato direttamente come input dell'$(i + 1)$-esimo circuito. Quindi, le configurazioni,

anziché essere rappresentate da registri di stato, sono i valori trasmessi sui fili che connettono le copie di M.

Si ricordi adesso ciò che l'algoritmo di riduzione con tempo polinomiale F deve fare. Dato un input x, esso deve calcolare un circuito $C = f(x)$ che sia soddisfattibile se e solo se esiste un certificato y tale che $A(x, y) = 1$. Quando F ottiene un input x, per prima cosa calcola $n = |x|$ e costruisce un circuito C' costituito da $T(n)$ copie di M. L'input di C' è una configurazione iniziale che corrisponde ad una computazione su $A(x,y)$, e l'output è la configurazione $c_{T(n)}$.

Il circuito $C = f(x)$ calcolato da F è ottenuto modificando leggermente C'. In primo luogo, gli input di C' corrispondenti a: il programma per A, il contatore del programma iniziale, l'input x, e lo stato iniziale della memoria sono collegati direttamente a questi valori noti. Per cui, l'unico input che rimane in ingresso al circuito corrisponde al certificato y. In secondo luogo, tutti gli output del circuito sono ignorati, eccetto il bit di $c_{T(n)}$ che corrisponde all'output di A. Il circuito C, così costruito, calcola $C(y) = A(x, y)$ per qualsiasi input y di lunghezza $O(n^k)$. L'algoritmo di riduzione F, quando gli si fornisce una stringa di input x, calcola un tale circuito C e lo restituisce come output.

Rimangono da dimostrare due proprietà. Per prima cosa si deve mostrare che F calcola in modo corretto una funzione di riduzione f. Cioè, si deve mostrare che C è soddisfattibile se e solo se esiste un certificato y tale che $A(x, y) = 1$. La seconda proprietà da dimostrare è la polinomialità del tempo di calcolo di F.

Per mostrare che F calcola correttamente una funzione di riduzione, si supponga che esista un certificato y di lunghezza $O(n^k)$ tale che $A(x, y) = 1$. Allora, se si applicano i bit di y agli input di C, l'output di C è $C(y) = A(x, y) = 1$. Quindi, se un certificato esiste, allora C è soddisfattibile. Per il viceversa, si supponga che C sia soddisfattibile; esiste quindi un input y per C tale che $C(y) = 1$, da cui si conclude che $A(x, y) = 1$; e quindi F calcola correttamente una funzione di riduzione.

Per completare la dimostrazione, è necessario mostrare soltanto che F è eseguito in tempo polinomiale rispetto a $n = |x|$. La prima osservazione che si può fare è che il numero di bit richiesti per rappresentare una configurazione è polinomiale in n. Il programma per A stesso ha una dimensione costante, indipendente dalla lunghezza del suo input x. La lunghezza dell'input x è n e la lunghezza del certificato y è $O(n^k)$. Poiché l'algoritmo è eseguito per al più $O(n^k)$ passi, l'ammontare della memoria di lavoro richiesta da A è anch'essa polinomiale in n. (Si assume che la memoria sia contigua; l'Esercizio 36.3-4 chiede di estendere l'argomentazione alla situazione in cui le locazioni accedute da A siano distribuite su una regione molto più grande di memoria e che il particolare modello di distribuzione possa essere differente per ciascun input x.)

Il circuito combinatorio M che realizza l'hardware del calcolatore ha una dimensione polinomiale rispetto alla lunghezza di una configurazione, che è polinomiale rispetto a $O(n^k)$ e quindi è polinomiale rispetto a n. (La maggior parte di questo circuito realizza la logica della memoria del sistema.) Il circuito C consiste di al più $t = O(n^k)$ copie di M e quindi ha una dimensione polinomiale rispetto a n. La costruzione di C da x può essere ottenuta in tempo polinomiale con l'algoritmo di riduzione F, poiché ciascun passo della costruzione impiega tempo polinomiale. ■

Il linguaggio SODDISFATTIBILITÀ-DI-CIRCUITI è perciò arduo almeno quanto qualsiasi linguaggio di NP e, poiché appartiene a NP, esso è NP-completo.

Figura 36.7 *La sequenza di configurazioni prodotte da un algoritmo A eseguito su un input x ed un certificato y. Ciascuna configurazione rappresenta lo stato del calcolatore per un passo di computazione e, oltre ad A, x e y, include il contatore del programma (PC), uno stato ausiliario della macchina e la memoria di lavoro. Eccetto che per il certificato y, la configurazione iniziale c_0 è costante. Ciascuna configurazione è trasformata nella successiva configurazione attraverso un circuito combinatorio M. L'output è uno specifico bit della memoria di lavoro.*

Teorema 36.7

Il problema della soddisfattibilità di circuiti è NP-completo.

Dimostrazione. Segue immediatamente dai Lemmi 36.5 e 36.6 e dalla definizione di NP-completezza. ■

Esercizi

36.3-1 Si mostri che la relazione $\leq_P$ è una relazione transitiva sui linguaggi. Cioè si mostri che se $L_1 \leq_P L_2$ e $L_2 \leq_P L_3$ allora $L_1 \leq_P L_3$.

36.3-2 Si dimostri che $L \leq_P \bar{L}$ se e solo se $\bar{L} \leq_P L$.

36.3-3 Si mostri che un assegnamento di soddisfattibilità può essere usato come certificato in una dimostrazione alternativa del lemma 36.5. Quale certificato consente la dimostrazione più semplice?

36.3-4 La dimostrazione del lemma 36.6 ipotizza che la memoria di lavoro per l'algoritmo A occupi una regione contigua di dimensione polinomiale. Nella dimostrazione, dove è impiegata questa ipotesi? Si spieghi che questa ipotesi non implica alcuna perdita di generalità.

36.3-5 Un linguaggio L è ***completo*** per una classe C di linguaggi rispetto alle riduzioni con tempo polinomiale, se $L \in C$ e $L' \leq_P L$ per ogni $L' \in C$. Si mostri che $\varnothing$ e $\{0, 1\}^*$ sono gli unici linguaggi in P che non sono completi per P rispetto alle riduzioni con tempo polinomiale.

36.3-6 Si mostri che L è completo per NP se e solo se $\bar{L}$ è completo per co-NP.

36.3-7 L'algoritmo di riduzione F della dimostrazione del lemma 36.6 costruisce il circuito $C = f(x)$ basandosi sulla conoscenza di A, x e k. Il professor Sartre osserva che la stringa x è un input di F: F conosce solo l'esistenza di A e k (dato che il linguaggio L appartiene a NP), ma non i loro valori effettivi. Quindi il professore conclude che F non può costruire il circuito C ed il linguaggio SODDISFATTIBILITÀ-DI-CIRCUITI non è necessariamente NP-arduo. Si spieghi l'errore nel ragionamento del professore.

36.4 Dimostrazioni di NP-completezza

La NP-completezza del problema della soddisfattibilità di circuiti si basa sulla dimostrazione diretta che $L \leq_P$ SODDISFATTIBILITÀ-DI-CIRCUITI per ogni linguaggio $L \in$ NP. In questo paragrafo, si mostrerà come provare che un linguaggio è NP-completo senza ridurre direttamente *ogni* linguaggio di NP a quel dato linguaggio. Si illustrerà questa metodologia dimostrando che diversi problemi di soddisfattibilità di formule sono NP-completi. Il paragrafo 36.5 fornisce molti altri esempi di questa metodologia.

Il seguente lemma è alla base del metodo per mostrare che un linguaggio è NP-completo.

Lemma 36.8

Se L è un linguaggio tale che $L' \leq_P L$ per qualche $L' \in$ NPC, allora L è NP-arduo. Inoltre, se $L \in$ NP, allora $L \in$ NPC.

Dimostrazione. Poiché L' è NP-completo, per tutti gli $L'' \in$ NP, si ha $L'' \leq_p L'$. Per ipotesi, $L' \leq_p L$ e quindi per transitività (si veda l'Esercizio 36.3-1), si ha che $L'' \leq_p L$, per cui L è NP-arduo. Se $L \in$ NP, si ha anche che $L \in$ NPC. ■

In altre parole, riducendo ad L un linguaggio NP-completo conosciuto L', implicitamente si riducono ad L tutti i linguaggi di NP. Quindi il lemma 36.8 fornisce un metodo per dimostrare che un linguaggio L è NP-completo:

1. Si dimostri che $L \in$ NP.
2. Si selezioni un linguaggio L' noto come linguaggio NP-completo.
3. Si descriva un algoritmo che calcoli una funzione f che fa corrispondere ad ogni istanza di L' una istanza di L.
4. Si dimostri che la funzione f è tale per cui $x \in L'$ se e solo se $f(x) \in L$, per tutti gli $x \in \{0, 1\}^*$.
5. Si dimostri che l'algoritmo che calcola f è eseguito in tempo polinomiale.

Questa metodologia di riduzione da un singolo linguaggio NP-completo noto è assai più semplice del procedimento più complesso di fornire le riduzioni da tutti i linguaggi di NP. La dimostrazione che SODDISFATTIBILITÀ-DI-CIRCUITI $\in$ NPC ha aperto un varco: la conoscenza che il problema della soddisfattibilità di circuiti è NP-completo permette di dimostrare molto più facilmente che altri problemi sono NP-completi. Inoltre, via via che si sviluppa un catalogo di linguaggi NP-completi, l'applicazione della metodologia sarà sempre più semplice.

Soddisfattibilità di formule

Si illustra la metodologia di riduzione fornendo la prova della NP-completezza del problema di determinare se una formula booleana, non un circuito, è soddisfattibile. Questo problema ha l'onore storico di essere stato il primo problema ad essere dimostrato NP-completo.

Si formalizza il problema della ***soddisfattibilità*** (***di una formula***) in termini del linguaggio SODDISFATTIBILITÀ come segue. Un'istanza di SODDISFATTIBILITÀ è una formula booleana ϕ composta di

1. variabili booleane: $x_1, x_2, \ldots$;
2. connettivi booleani: qualsiasi funzione booleana con uno o due argomenti e un output, come $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implica), $\leftrightarrow$ (se e solo se); e
3. parentesi.

Come nei circuiti combinatori, un ***assegnamento di verità*** per una formula booleana ϕ è un insieme di valori per le variabili di ϕ, ed un ***assegnamento di soddisfattibilità*** è un assegnamento di verità che implica che la formula sia uguale ad 1. Una formula con un assegnamento di soddifattibilità si dice ***soddisfattibile***. Il problema della soddisfattibilità chiede se una data formula booleana è soddisfattibile; in termini di linguaggi formali,

SODDISFATTIBILITÀ = $\{\langle\phi\rangle$: ϕ è una formula booleana soddisfattibile$\}$.

Per esempio, la formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

ha l'assegnamento di soddisfattibilità $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, poiché

$$\begin{aligned}\phi &= ((0 \rightarrow 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1,\end{aligned} \tag{36.2}$$

per cui questa formula ϕ appartiene a SODDISFATTIBILITÀ.

L'algoritmo più naturale per determinare se una generica formula booleana è soddisfattibile non è eseguito in tempo polinomiale. Esistono 2^n possibili assegnamenti per una formula ϕ con n variabili. Se la lunghezza di $\langle\phi\rangle$ è polinomiale rispetto ad n, allora la verifica di tutti gli assegnamenti richiede un tempo super polinomiale. Il seguente teorema mostra come sia inverosimile l'esistenza di un algoritmo con tempo polinomiale.

Teorema 36.9

La soddisfattibilità delle formule booleane è un problema NP-completo.

Dimostrazione. Per prima cosa si mostrerà che SODDISFATTIBILITÀ $\in$ NP. Poi si mostrerà che SODDISFATTIBILITÀ-DI-CIRCUITI $\leq_p$ SODDISFATTIBILITÀ e ciò, per il lemma 36.8, dimostrerà il teorema.

Per mostrare che SODDISFATTIBILITÀ appartiene a NP, si mostra che un certificato costituito da un assegnamento di soddisfattibilità per una data formula ϕ può essere verificato in tempo polinomiale. Questo algoritmo di verifica sostituisce semplicemente ciascuna variabile della formula con il suo corrispondente valore e quindi valuta l'espressione, facendo così quello che si è fatto per l'equazione (36.2) di cui sopra. Questo compito è facilmente eseguibile in tempo polinomiale. Se l'espressione vale 1, allora la formula è soddisfattibile. Quindi vale la prima condizione del lemma 36.8 per la NP-completezza.

Per dimostrare che SODDISFATTIBILITÀ è NP-arduo, si mostra che SODDISFATTIBILITÀ-DI-CIRCUITI $\leq_p$ SODDISFATTIBILITÀ. In altre parole, qualsiasi istanza della soddisfattibilità di circuiti può essere ridotta in tempo polinomiale ad un'istanza della soddisfattibilità di formule. Si può usare l'induzione per esprimere qualsiasi circuito combinatorio booleano come una formula booleana. Semplicemente, si considera la porta che produce l'output ed induttivamente si esprime ciascun ingresso di tale porta come una formula. La formula per il circuito è quindi ottenuta scrivendo un'espressione che applica la funzione della porta alle formule che ne costituiscono l'input.

Sfortunatamente, questo metodo banale non costituisce una riduzione in tempo polinomiale. Sottoformule condivise possono far crescere in modo esponenziale la dimensione della formula generata (si veda l'Esercizio 36.4-1). Quindi l'algoritmo di riduzione deve essere in qualche modo più astuto.

La figura 36.8 illustra l'idea di base della riduzione da SODDISFATTIBILITÀ-DI-CIRCUITI a SODDISFATTIBILITÀ per il circuito di figura 36.6(a). Per ciascuno dei fili x_i nel circuito C, la formula ϕ ha una variabile x_i. La specifica operazione di una porta può adesso

essere espressa come una formula sulle variabili dei suoi fili incidenti. Per esempio, l'operazione della porta AND di output è $x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)$.

La formula ϕ prodotta dall'algoritmo di riduzione è un AND delle variabili di output del circuito con la congiunzione delle clausole che descrivono le operazioni di ciascuna porta. Per il circuito della figura, la formula è

$$\begin{aligned}\phi = x_{10} \;&\wedge\; (x_4 \leftrightarrow \neg x_3)\\ &\wedge\; (x_5 \leftrightarrow (x_1 \vee x_2))\\ &\wedge\; (x_6 \leftrightarrow \neg x_4)\\ &\wedge\; (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))\\ &\wedge\; (x_8 \leftrightarrow (x_5 \vee x_6))\\ &\wedge\; (x_9 \leftrightarrow (x_6 \vee x_7))\\ &\wedge\; (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) .\end{aligned}$$

Dato un circuito C, è semplice produrre una tale formula ϕ in tempo polinomiale.

Perché il circuito C è soddisfattibile esattamente quando lo è la formula ϕ? Se C ha un assegnamento di soddisfattibilità, ciascun filo del circuito ha un valore ben definito, e l'output del circuito è 1. Perciò, l'assegnamento dei valori dei fili alle variabili di ϕ comporta che ciascuna clausola in ϕ valga 1 e quindi la loro congiunzione vale 1. Viceversa, se c'è un assegnamento che implica che ϕ valga 1, allora il circuito C è soddisfattibile seguendo un ragionamento analogo. Quindi, si è mostrato che SODDISFATTIBILITÀ-DI-CIRCUITI $\leq_p$ SODDISFATTIBILITÀ, completando così la dimostrazione. ■

3-soddisfattibilità in forma normale congiuntiva

Molti problemi possono essere dimostrati NP-completi per riduzione della soddisfattibilità di formule. L'algoritmo di riduzione deve gestire, però, qualsiasi formula in input e questo può condurre ad un enorme numero di casi che devono essere considerati. Perciò, è spesso desiderabile effettuare la riduzione partendo da un insieme ristretto di formule booleane, in modo che ci siano da considerare meno casi. Naturalmente non si deve restringere il linguaggio così tanto da renderlo risolvibile in tempo polinomiale. Un linguaggio conveniente è la 3-soddisfattibilità in forma normale congiuntiva, o 3-SODDISFATTIBILITÀ-FNC.

La 3-soddisfattibilità in forma normale congiuntiva si definisce usando i seguenti termini. Una *lettera* in una formula booleana è un'occorrenza di una variabile o della sua negazione. Una formula booleana è ***in forma normale congiuntiva***, o ***FNC***, se è espressa come un AND di clausole, ciascuna delle quali è un OR di una o più lettere. Una formula booleana è in ***forma normale 3-congiuntiva***, o ***3-FNC***, se ciascuna clausola ha esattamente tre lettere distinte.

Per esempio, la formula booleana

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

è in 3-FNC. La prima delle sue tre clausole è $(x_1 \vee \neg x_1 \vee \neg x_2)$, e contiene le tre lettere x_1, $\neg x_1$, e $\neg x_2$.

Il problema 3-SODDISFATTIBILITÀ-FNC consiste nello stabilire se una data formula booleana ϕ in 3-FNC è soddisfattibile. Il seguente teorema mostra che è inverosimile che

Figura 36.8 *La riduzione della soddisfattibilità di circuiti alla soddisfattibilità di formule. La formula prodotta dall'algoritmo di riduzione ha una variabile per ogni filo del circuito.*

esista un algoritmo che possa determinare in tempo polinomiale la soddisfattibilità di formule booleane anche se sono espresse in questa forma normale più semplice.

Teorema 36.10

La soddisfattibilità di formule booleane espresse in forma normale 3-congiuntiva è un problema NP-completo.

Dimostrazione. Le argomentazioni usate nella dimostrazione del Teorema 36.9 per mostrare che SODDISFATTIBILITÀ $\in$ NP possono essere applicate nello stesso modo per mostrare che anche 3-SODDISFATTIBILITÀ-FNC $\in$ NP. Quindi, si deve dimostrare solo che 3-SODDISFATTIBILITÀ-FNC è NP-arduo. Si prova ciò mostrando che SODDISFATTIBILITÀ $\leq_p$ 3-SODDISFATTIBILITÀ-FNC, da cui segue la dimostrazione in base al lemma 36.8.

L'algoritmo di riduzione può essere suddiviso in tre passi fondamentali. Ciascun passo progressivamente trasforma la formula di input ϕ nella sua corrispondente rappresentazione in forma normale 3-congiuntiva.

Il primo passo è simile a quello usato nel Teorema 36.9 per dimostrare che SODDISFATTIBILITÀ-DI-CIRCUITI $\leq_p$ SODDISFATTIBILITÀ. Per prima cosa, si codifica la formula ϕ in input con un albero binario di "parsing", in cui le lettere sono le foglie ed i connettivi sono i nodi interni. La figura 36.9 mostra l'albero di parsing della formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg ((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2. \tag{36.3}$$

Se la formula in input contiene operazioni di AND e OR su più di due lettere, allora si può usare la proprietà associativa per inserire parentesi nella formula in modo che ogni nodo interno abbia 1 o 2 figli. L'albero binario di parsing può adesso essere visto come un circuito per calcolare la funzione.

Imitando la riduzione usata nella dimostrazione del Teorema 36.9, si introduce una variabile y_i per l'output di ciascun nodo interno. Poi si riscrive la formula originaria ϕ come un AND tra la variabile della radice ed una congiunzione di clausole che descrivono le operazioni di ciascun nodo. Per esempio, per la formula (36.3) l'espressione risultante è

$$\begin{aligned}\phi' = y_1 \;&\wedge\; (y_1 \leftrightarrow (y_2 \wedge \neg x_2))\\ &\wedge\; (y_2 \leftrightarrow (y_3 \vee y_4))\\ &\wedge\; (y_3 \leftrightarrow (x_1 \rightarrow x_2))\end{aligned}$$

Figura 36.9 *L'albero corrispondente alla formula* $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

$$\begin{aligned} &\wedge \quad (y_4 \leftrightarrow \neg y_5) \\ &\wedge \quad (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge \quad (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \,. \end{aligned}$$

Si osservi che la formula ϕ' così ottenuta è una congiunzione di clausole ϕ'_i, ciascuna delle quali ha al più 3 lettere. Il solo requisito ulteriore da rispettare è che ciascuna clausola sia un OR di lettere.

Il secondo passo di riduzione trasforma ciascuna clausola ϕ'_i in forma normale congiuntiva. Si costruisce una tabella di verità per ϕ'_i valutando tutti i possibili assegnamenti delle sue variabili. Ciascuna riga della tabella di verità consiste di un possibile assegnamento alle variabili della clausola, insieme al valore che ha la clausola in base a quell'assegnamento. Usando gli elementi della tabella di verità che valgono 0, si determina una formula in ***forma normale disgiuntiva*** (o ***FND***) – un OR di AND – che è equivalente a $\neg \phi'_i$. Si trasforma questa formula in una formula ϕ''_i in FNC, usando le leggi di DeMorgan (5.2) per complementare tutte le lettere e cambiare gli OR in AND e gli AND in OR.

Nell'esempio, si trasforma la clausola $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ in FNC come segue. La tabella di verità di ϕ'_1 è data nella figura 36.10. La formula in FND equivalente a $\neg \phi'_1$ è

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2) \,.$$

Applicando le leggi di DeMorgan, si ha la formula in FNC

$$\phi''_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2) \,,$$

che è equivalente alla clausola originaria ϕ'_1.

Ciascuna clausola ϕ'_i della formula ϕ' è stata così trasformata in una formula ϕ''_i in FNC, e quindi ϕ' è equivalente alla formula ϕ'' in FNC, composta dalla congiunzione dei ϕ''_i. Inoltre, ciascuna clausola di ϕ'' ha al più tre lettere.

Il terzo ed ultimo passo di riduzione trasforma ulteriormente la formula in modo che ciascuna clausola abbia *esattamente* 3 lettere distinte. La formula finale ϕ''' in 3-FNC, è ottenuta dalle clausole della formula di ϕ'' che è in FNC. Questo passo usa due variabili ausiliarie che si chiameranno p e q. Per ciascuna clausola C_i di ϕ'', si includono in ϕ''' le seguenti clausole:

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 *La tabella di verità per la clausola* $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

- Se C_i ha 3 lettere distinte, allora si include direttamente C_i come clausola di ϕ'''.
- Se C_i ha 2 lettere distinte, cioè, se $C_i = (l_1 \vee l_2)$, dove l_1 e l_2 sono lettere, allora si includono $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ come clausole di ϕ'''. Le lettere p e $\neg p$, servono semplicemente per verificare il requisito di avere esattamente 3 lettere distinte per ogni clausola: $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ è equivalente a $(l_1 \vee l_2)$ sia che $p = 0$ sia che $p = 1$.
- Se C_i ha solo una lettera l, allora si includono $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ come clausole di ϕ'''. Si noti che per qualsiasi assegnamento a p e q la congiunzione di queste quattro clausole è sempre uguale al valore di l.

Esaminando i passi della trasformazione, si può vedere che la formula ϕ''' in 3-FNC è soddisfattibile se e solo se ϕ è soddisfattibile. La costruzione di ϕ' da ϕ introduce al più 1 variabile ed 1 clausola per ogni connettivo in ϕ. La costruzione di ϕ'' da ϕ' può introdurre al più 8 clausole in ϕ'' per ogni clausola di ϕ', poiché ciascuna clausola in ϕ' ha al più 3 variabili, e la tabella di verità per ciascuna clausola ha al più $2^3 = 8$ righe. La costruzione di ϕ''' da ϕ'' introduce al più 4 clausole in ϕ''' per ogni clausola in ϕ''. Quindi la dimensione della formula risultante ϕ''' è polinomiale rispetto alla lunghezza della formula originale ϕ. Ciascuna costruzione può essere facilmente ottenuta in tempo polinomiale. ■

Esercizi

36.4-1 Si consideri la riduzione banale (non di tempo polinomiale) accennata nella dimostrazione del Teorema 36.9. Si descriva un circuito di dimensione n che, se trasformato in una formula con questo metodo, produce una formula la cui dimensione è esponenziale in n.

36.4-2 Si mostri la formula in 3-FNC che risulta quando si usa il metodo del Teorema 36.10 sulla formula (36.3).

36.4-3 Il professor Jagger propone di mostrare che SODDISFATTIBILITÀ $\leq_P$ 3-SODDISFATTIBILITÀ-FNC usando soltanto la tecnica della tabella di verità della dimostrazione del Teorema 36.10 e nessun altro passo. Cioè, il professore propone di: prendere la formula booleana ϕ, determinare una tabella di verità per le sue variabili, derivare dalla tabella di verità una formula in 3-FND che sia equivalente a $\neg\phi$, e poi negare ed applicare le leggi di DeMorgan producendo una formula in 3-FNC equivalente a ϕ. Si mostri che questa strategia non produce una riduzione in tempo polinomiale.

36.4-4 Si mostri che il problema di determinare se una formula booleana è una tautologia è completo per co-NP. (*Suggerimento*: si veda l'esercizio 36.3-6.)

36.4-5 Si mostri che il problema della determinazione della soddisfattibilità di formule in forma normale disgiuntiva è risolvibile in tempo polinomiale.

36.4-6 Si supponga che qualcuno fornisca un algoritmo di tempo polinomiale per decidere la soddisfattibilità di formule. Si descriva come usare questo algoritmo per trovare gli assegnamenti di soddisfattibilità in tempo polinomiale.

36.4-7 Sia 2-SODDISFATTIBILITÀ-FNC l'insieme delle formule soddisfattibili in FNC con esattamente 2 lettere per clausola. Si mostri che 2-SODDISFATTIBILITÀ-FNC $\in$ P. Si determini un algoritmo che sia il più efficiente possibile. (*Suggerimento*: si osservi che $x \vee y$ è equivalente a $\neg x \rightarrow y$. Si riduca 2-SODDISFATTIBILITÀ-FNC ad un problema su un grafo orientato che è risolvibile in modo efficiente.)

36.5 Problemi NP-completi

I problemi NP-completi si ritrovano in diverse aree: logica booleana, grafi, aritmetica, progettazione di reti, insiemi e partizioni, memoria e ricerca, sequenzializzazione e pianificazione, programmazione matematica, algebra e teoria dei numeri, giochi e puzzle, teoria dei linguaggi e degli automi, ottimizzazione di programmi ed altro. In questo paragrafo, si userà la metodologia di riduzione per fornire la dimostrazione di NP-completezza per molti problemi presi dalla teoria dei grafi e dal partizionamento di insiemi.

La figura 36.11 schematizza la struttura delle dimostrazioni di NP-completezza di questo paragrafo e del paragrafo 36.4. Ciascun linguaggio della figura è stabilito essere NP-completo per riduzione dal linguaggio che lo punta. Alla radice si trova SODDISFATTIBILITÀ-DI-CIRCUITI, che è stato dimostrato NP-completo nel Teorema 36.7.

36.5.1 Il problema della cricca

Una *cricca* in un grafo non orientato $G = (V, E)$ è un sottoinsieme $V' \subseteq V$ di vertici tale che ogni coppia di suoi vertici è connessa da un arco in E. In altre parole, una cricca è un sottografo completo di G. La ***dimensione*** di una cricca è il numero di vertici che contiene. Il ***problema della cricca*** è il problema di ottimizzazione: trovare in un grafo una cricca di dimensione massima. Come problema di decisione, si chiede semplicemente se esiste nel grafo una cricca di una data dimensione k. La definizione formale è

CRICCA = $\{\langle G, k\rangle$: G è un grafo con una cricca di dimensione $k\}$.

Un algoritmo immediato per determinare se un grafo $G = (V, E)$ con $|V|$ vertici ha una cricca di dimensione k è elencare tutti i k-sottoinsiemi di V, e considerarli uno per uno per vedere

Figura 36.11 *La struttura delle dimostrazioni di* NP-*completezza dei paragrafi* 36.4 *e* 36.5. *Tutte le dimostrazioni, sostanzialmente, sono ottenute per riduzione dalla* NP-*completezza di* SODDISFATTIBILITÀ-DI-CIRCUITI.

se formano una cricca. Il tempo di esecuzione di questo algoritmo è $\Theta(k^2 \binom{|V|}{k})$, che è polinomiale se k è costante. In generale, tuttavia, k potrebbe essere proporzionale a $|V|$, nel qual caso l'algoritmo sarebbe eseguito in tempo super polinomiale. È inverosimile che esista un algoritmo efficiente per il problema della cricca.

Teorema 36.11

Il problema della cricca è NP-completo.

Dimostrazione. Per mostrare che CRICCA $\in$ NP, per un dato grafo $G = (V, E)$, si usa l'insieme $V' \subseteq V$ dei vertici della cricca come certificato per G. La verifica che V' sia una cricca può essere ottenuta in tempo polinomiale verificando se, per ogni coppia $u, v \in V'$, l'arco (u, v) appartiene ad E.

Si mostra che il problema della cricca è NP-arduo dimostrando che 3-SODDISFATTIBILITÀ-FNC $\leq_p$ CRICCA. Che sia possibile dimostrare questo risultato è a prima vista sorprendente, poiché le formule logiche non sembrano avere molto in comune con i grafi.

L'algoritmo di riduzione comincia con un'istanza di 3-SODDISFATTIBILITÀ-FNC. Sia $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ una formula booleana in 3-FNC con k clausole. Per $r = 1, 2, \ldots, k$, ciascuna clausola C_r ha esattamente 3 lettere distinte l_1^r, l_2^r e l_3^r. Si determinerà un grafo G tale che ϕ è soddisfattibile se e solo se G ha una cricca di dimensione k.

Il grafo $G = (V, E)$ è costruito come segue. Per ciascuna clausola $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, si pone una tripla di vertici v_1^r, v_2^r e v_3^r in V. Si pone un arco tra due vertici v_i^r e v_j^s se valgono contemporaneamente le seguenti condizioni:

- v_i^r e v_j^s sono in triple differenti, cioè $r \neq s$, e
- le loro corrispondenti lettere sono ***compatibili***, cioè l_i^r non è la negazione di l_j^s.

Questo grafo può essere facilmente calcolato a partire da ϕ in tempo polinomiale. Come esempio di questa costruzione, se si ha

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3),$$

allora G è il grafo mostrato nella figura 36.12.

Si deve mostrare che questa trasformazione di ϕ in G è una riduzione. Per prima cosa si supponga che ϕ abbia un assegnamento soddisfacente. Allora, ciascuna clausola C_r contiene almeno una lettera l_i^r a cui è assegnato 1, e ciascuna di queste lettere corrisponde ad un vertice v_i^r. Prendendo da ciascuna clausola una di tali lettere "vero", si ottiene un insieme di vertici V'. Si sostiene che V' è una cricca. Per qualsiasi coppia di vertici v_i^r e $v_j^s \in V'$, se $r \neq s$, entrambi le corrispondenti lettere l_i^r e l_j^s hanno valore 1 per l'assegnamento soddisfacente dato e quindi le lettere non possono essere complementari. Quindi, per la costruzione di G, l'arco (v_i^r, v_j^s) appartiene ad E.

Viceversa, si supponga che G abbia una cricca di dimensione k. Nessun arco in G connette due vertici della stessa tripla, e così V' contiene esattamente un vertice per tripla. Si può assegnare 1 a ciascuna lettera l_i^r tale che $v_i^r \in V'$ senza temere di assegnare 1 sia ad una lettera che al suo complemento, poiché G non contiene archi tra lettere non consistenti. Ciascuna clausola è soddisfatta, e così ϕ è soddisfatta. (Alle variabili che non corrispondono ad alcun vertice della cricca può essere assegnato un valore arbitrario.) ■

Nell'esempio della figura 36.12, un assegnamento soddisfacente per ϕ è $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. Un cricca corrispondente di dimensione $k = 3$ è costituita dai vertici corrispondenti a $\neg x_2$ della prima clausola, x_3 della seconda clausola, ed x_3 della terza clausola.

36.5.2 Il problema della copertura di vertici

Una ***copertura di vertici*** di un grafo non orientato $G = (V, E)$ è un sottoinsieme $V' \subseteq V$ tale che se $(u, v) \in E$, allora $u \in V'$ o $v \in V'$ (o entrambi). Cioè, ciascun vertice "copre" i suoi archi incidenti, ed una copertura di vertici per G è un insieme di vertici che copre tutti gli archi in G. La ***dimensione*** di una copertura di vertici è il numero di vertici in essa. Per esempio, il grafo nella figura 36.13(b) ha una copertura di vertici $\{w, z\}$ di dimensione 2.

Il ***problema della copertura di vertici*** è di trovare in un grafo dato una copertura di vertici con la dimensione minima. La riformulazione di questo problema di ottimizzazione come un problema di decisione, richiede di determinare se un grafo ha una copertura di vertici di una dimensione k data. In termini di linguaggio, si definisce

COPERTURA-DI-VERTICI = $\{\langle G, k\rangle$: il grafo G ha una copertura di dimensione $k\}$.

Il seguente teorema mostra che questo problema è NP-completo.

Teorema 36.12

Il problema della copertura di vertici è NP-completo.

Dimostrazione. Si mostra prima che COPERTURA-DI-VERTICI $\in$ NP. Si supponga che sia dato un grafo $G = (V, E)$ ed un intero k. Il certificato che si sceglie è la copertura $V' \subseteq V$ stessa. L'algoritmo di verifica conferma che $|V'| = k$ e poi verifica, per ciascun arco $(u, v) \in E$, se $u \in V'$ o $v \in V'$. Questa verifica può essere comodamente eseguita in tempo polinomiale.

Si dimostra che il problema della copertura di vertici è NP-completo, mostrando che CRICCA $\leq_P$ COPERTURA-DI-VERTICI. Questa riduzione è basata sulla nozione di "complemento" di un grafo. Dato un grafo non orientato $G = (V, E)$, si definisce il ***complemento*** di G, $\bar{G} = (V, \bar{E})$, dove $\bar{E} = \{(u, v) : (u, v) \notin E\}$. In altre parole, $\bar{G}$ è il grafo

Figura 36.12 *Il grafo G derivato dalla formula in* 3-FNC $\phi = C_1 \wedge C_2 \wedge C_3$, *dove* $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$ *e* $C_3 = (x_1 \vee x_2 \vee x_3)$, *nella riduzione di* 3-SODDISFATTIBILITÀ-FNC *a* CRICCA. *Un assegnamento di soddisfattibilità della formula è* $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. *Questo assegnamento soddisfa* C_1 *con* $x_2 = 0$ *e soddisfa* C_2 *e* C_3 *con* $x_3 = 1$, *e corrisponde alla cricca con i vertici in grigio chiaro.*

contenente esattamente quegli archi che non sono in G. La figura 36.13 mostra un grafo ed il suo complemento ed illustra la riduzione da CRICCA a COPERTURA-DI-VERTICI.

L'algoritmo di riduzione prende in input un'istanza $\langle G, k\rangle$ del problema della cricca e calcola il complemento $\bar{G}$, che può essere facilmente ottenuto in tempo polinomiale. L'output dell'algoritmo di riduzione è l'istanza $\langle \bar{G}, |V| - k\rangle$ del problema della copertura di vertici. Per completare la dimostrazione si fa vedere che questa trasformazione è davvero una riduzione: il grafo G ha una cricca di dimensione k se e solo se il grafo $\bar{G}$ ha una copertura di vertici di dimensione $|V| - k$.

Si supponga che G abbia una cricca $V' \subseteq V$ con $|V'| = k$. Sosteniamo che $V - V'$ è una copertura di vertici di $\bar{G}$. Sia (u, v) un qualsiasi arco di $\bar{E}$. Allora $(u, v) \notin E$, ciò implica che almeno uno tra u e v non appartenga a V', poiché ogni coppia di vertici di V' è connessa da un arco in E. Equivalentemente, almeno uno tra u e v appartiene a $V - V'$, il che significa che l'arco (u, v) è coperto da $V - V'$. Poiché (u, v) è stato scelto in modo arbitrario in $\bar{E}$, ogni arco di $\bar{E}$ è coperto da un vertice di $V - V'$. Quindi, l'insieme $V - V'$, che ha dimensione $|V| - k$, è una copertura di vertici di $\bar{G}$.

Viceversa, si supponga che $\bar{G}$ abbia una copertura di vertici $V' \subseteq V$, dove $|V'| = |V| - k$. Allora, per tutti gli u e $v \in V$, se $(u, v) \in \bar{E}$, allora $u \in V'$ oppure $v \in V'$ o valgono entrambe le appartenenze. La contronominale di questa implicazione asserisce che per tutti gli u e $v \in V$, se $u \notin V'$ e $v \notin V'$, allora $(u, v) \in E$. In altre parole, $V - V'$ è una cricca, ed ha dimensione $|V| - |V'| = k$. ■

Poiché COPERTURA-DI-VERTICI è NP-completo, non ci si aspetta di trovare un algoritmo di tempo polinomiale per la ricerca della copertura di vertici di dimensione minima. Il paragrafo 37.1 presenta, tuttavia, un "algoritmo approssimato" con tempo polinomiale, che produce soluzioni "approssimate" per il problema della copertura di vertici. La dimensione di una copertura di vertici prodotta dall'algoritmo è al più doppia di quella di una copertura di vertici di dimensione minima.

Figura 36.13 *La riduzione di* CRICCA a COPERTURA-DI-VERTICI. **(a)** *Un grafo non orientato G = (V, E) con una cricca V' = {u, v, x, y}.* **(b)** *Il grafo $\overline{G}$ prodotto dall'algoritmo di riduzione che ha copertura di vertici V − V' = {w, z}.*

Quindi, non si deve abbandonare la speranza malgrado il problema sia NP-completo. Ci può essere un algoritmo approssimato di tempo polinomiale che ottiene soluzioni quasi ottime, anche se la ricerca di una soluzione ottima è NP-completa. Il Capitolo 37 fornisce diversi algoritmi approssimati per problemi NP-completi.

36.5.3 Il problema della somma di sottoinsieme

Il prossimo problema NP-completo che si considera è aritmetico. Nel ***problema della somma di sottoinsieme***, sono dati un insieme finito $S \subset \mathbb{N}$ ed un ***obiettivo*** $t \in \mathbb{N}$. Si chiede se esiste un sottoinsieme $S' \subseteq S$, i cui elementi danno come somma t. Per esempio, se $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ e $t = 3754$, allora il sottoinsieme $S' = \{1, 16, 64, 256, 1040, 1093, 1284\}$ è una soluzione.

Come al solito, si definisce il problema come un linguaggio:

SOMMA-DI-SOTTOINSIEME = $\{\langle S, t\rangle$: esiste un sottoinsieme $S' \subseteq S$ tale che $t = \sum_{s \in S'} s\}$.

Come per qualsiasi problema aritmetico, è importante ricordare che la codifica standard assume che input interi siano codificati in binario. Tenendo presente questa ipotesi, si può mostrare che è inverosimile avere un algoritmo veloce per il problema della somma di sottoinsieme.

Teorema 36.13

Il problema della somma di sottoinsieme è NP-completo.

Dimostrazione. Per mostrare che SOMMA-DI-SOTTOINSIEME è in NP, per un'istanza $\langle S, t\rangle$ del problema, si consideri il sottoinsieme S' come certificato. La verifica che $t = \sum_{s \in S'} s$ può essere ottenuta con un algoritmo di verifica in tempo polinomiale.

Si mostra adesso che COPERTURA-DI-VERTICI $\leq_p$ SOMMA-DI-SOTTOINSIEME. Data un'istanza $\langle G, k\rangle$ del problema di copertura di vertici, l'algoritmo di riduzione costruisce un'istanza $\langle S, t\rangle$ del problema della somma di sottoinsieme, tale che G ha una copertura di vertici con dimensione k se e solo se c'è un sottoinsieme di S la cui somma vale esattamente t.

Al cuore della riduzione c'è una rappresentazione con matrice di incidenza di G. Sia $G = (V, E)$ un grafo non orientato e sia $V = \{v_0, v_1, \ldots, v_{|V|-1}\}$ ed $E = \{e_0, e_1, \ldots, e_{|E|-1}\}$. La ***matrice di incidenza*** di G è una matrice $|V| \times |E|$ $B = (b_{ij})$ tale che

$$b_{ij} = \begin{cases} 1 & \text{se l'arco } e_j \text{ è incidente sul vertice } v_i\ , \\ 0 & \text{altrimenti}\ . \end{cases}$$

Per esempio, la figura 36.14(b) mostra la matrice di incidenza per il grafo non orientato della figura 36.14(a). La matrice di incidenza è mostrata con gli archi di indice più basso sulla destra, piuttosto che sulla sinistra come si fa di solito, ciò per semplificare le formule per i numeri di S.

Dati un grafo G ed un intero k, l'algoritmo di riduzione calcola un insieme S di numeri ed un intero t. Per comprendere come funziona l'algoritmo di riduzione, si rappresentino i numeri in "base 4 modificata". Le $|E|$ cifre di ordine più basso di un numero saranno rappresentate in base 4, ma la cifra di ordine più alto può essere grande quanto k. L'insieme dei numeri è costruito in modo tale che nessun riporto può essere propagato dalle cifre di ordine più basso a quelle di ordine più alto.

L'insieme S consiste di due tipi di numeri, corrispondenti rispettivamente ai vertici e agli archi. Per ogni vertice $v_i \in V$, si crea un intero positivo x_i la cui rappresentazione in base 4 modificata consiste di un 1 seguito da $|E|$ cifre. Le cifre corrispondono alla riga di v_i nella matrice di incidenza $B = (b_{ij})$ di G, come illustrato nella figura 36.14(c). Formalmente, per $i = 0, 1, \ldots, |V| - 1$,

$$x_i = 4^{|E|} + \sum_{j=0}^{|E|-1} b_{ij} 4^j\ .$$

Figura 36.14 *La riduzione del problema della copertura di vertici al problema della somma di sottoinsieme.* **(a)** *Un grafo non orientato G. Una copertura di vertici $\{v_1, v_3, v_4\}$ di dimensione 3 è colorata in grigio chiaro.* **(b)** *La matrice di incidenza corrispondente. Le righe in grigio chiaro corrispondono ai vertici della copertura della parte* (a). *Ciascun arco e_j ha un 1 in almeno una riga in grigio chiaro.* **(c)** *La corrispondente istanza della somma di sottoinsieme. La porzione dentro il quadrato è la matrice di incidenza. Qui, la copertura di vertici $\{v_1, v_3, v_4\}$ di dimensione 3 corrisponde al sottoinsieme in grigio chiaro* {1, 16, 64, 256, 1040, 1093, 1284}, *la cui somma raggiunge* 3754.

Per ogni arco $e_j \in E$, si crea un intero positivo y_j che è semplicemente una riga della matrice "identità" di incidenza. (La matrice identità di incidenza è $|E| \times |E|$ con gli 1 solo nelle posizioni della diagonale.) Formalmente, per $j = 0, 1, \ldots, |E| - 1$,

$$y_j = 4^j .$$

La prima cifra della somma obiettivo t è k e tutte le $|E|$ cifre di ordine più basso sono sempre uguali a 2. Formalmente

$$t = k4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^j .$$

Tutti questi numeri hanno una dimensione polinomiale quando sono rappresentati in binario. La riduzione può quindi essere eseguita in tempo polinomiale attraverso la manipolazione dei bit della matrice di incidenza.

Si deve adesso mostrare che il grafo G ha una copertura di vertici di dimensione k se e solo se esiste un sottoinsieme $S' \subseteq S$ la cui somma è t. Per prima cosa si supponga che G abbia una copertura di vertici $V' \subseteq V$ di dimensione k. Sia $V' = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$, e si definisca S' come

$$S' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \cup \{y_j : e_j \text{ è incidente esattamente su un solo vertice in } V'\} .$$

Per verificare che $\sum_{s \in S'} s = t$, si osservi che la somma dei k 1 iniziali degli $x_{i_m} \in S'$ fornisce la cifra iniziale k della rappresentazione di t in base 4 modificata. Per ottenere le cifre di ordine più basso in t, ciascuna delle quali è un 2, si considerino una dopo l'altra le posizioni corrispondenti agli archi e_j. Dato che V' è una copertura di vertici, e_j è incidente su almeno un vertice di V'. Quindi, per ciascun arco e_j, c'è almeno un $x_{i_m} \in S'$ con un 1 nella j-esima posizione. Se e_j è incidente su due vertici di V', allora entrambi contribuiscono per un 1 alla somma nella j-esima posizione. La j-esima cifra di y_j non contribuisce per niente, poiché e_j è incidente su due vertici, il che implica che $y_j \notin S'$. Quindi, in questo caso, la somma di S' produce un 2 nella j-esima posizione di t. Nell'altro caso – quando e_j è incidente su esattamente un vertice di V' – si ha $y_j \in S'$, ed il vertice incidente e y_j contribuiscono ciascuno con un 1 alla somma della j-esima cifra di t, producendo così ancora un 2. Quindi, S' è una soluzione della somma di sottoinsieme dell'istanza S.

Adesso, si supponga che ci sia un sottoinsieme $S' \subseteq S$ la cui somma è t. Sia $S' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_k}\} \cup \{y_{j_1}, y_{j_2}, \ldots, y_{j_k}\}$. Sosteniamo che $m = k$ e che $V' = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ è una copertura di vertici per G. Per dimostrare questa affermazione osserviamo che, per ogni arco $e_j \in E$, l'insieme S contiene tre interi aventi 1 nella posizione e_j: due sono relativi ai due vertici incidenti su e_j, ed uno viene da y_j. Dato che si sta operando con le rappresentazioni in base 4 modificata, non ci sono riporti dalla posizione e_j alla posizione e_{j+1}. Quindi, per ciascuna delle $|E|$ posizioni di ordine più basso in t, almeno una ed al più due x_i devono contribuire alla somma. Poiché almeno una x_i di ciascun arco contribuisce alla somma, si verifica che V' è una copertura di vertici. Per verificare che $m = k$, e quindi che V' è una copertura di vertici di dimensione k, si osservi che l'unico modo perché sia raggiunta la cifra k iniziale nell'obiettivo t è che nella somma siano inclusi esattamente k degli x_i. ■

Nella figura 36.14, la copertura di vertici $V' = \{v_1, v_4, v_5\}$ corrisponde al sottoinsieme $S' = \{x_1, x_3, x_4, y_6, y_2, y_3, y_4\}$. Tutti gli y_j sono inclusi in S' ad eccezione di y_1, che è incidente su due vertici di V'.

Figura 36.15 (a) *La sottostruttura A, usata nella riduzione da* 3-SODDISFATTIBILITÀ-FNC *a* CICLO-HAMILTONIANO. (b)-(c) *Se A è un sottografo di un qualche grafo G che contiene un ciclo hamiltoniano e le uniche connessioni da A al resto di G sono attraverso i vertici a, a', b e b', allora gli archi in grigio rappresentano gli unici due modi possibili per i quali il ciclo hamiltoniano può attraversare gli archi del sottografo* A. (d) *Una rappresentazione compatta della sottostruttura A.*

36.5.4 Il problema del ciclo hamiltoniano

Si riesamina adesso il problema del ciclo hamiltoniano definito nel paragrafo 36.2.

Teorema 36.14

Il problema del ciclo hamiltoniano è NP-completo.

Dimostrazione. Si mostra prima che CICLO-HAMILTONIANO appartiene ad NP. Dato un grafo $G = (V, E)$ il certificato da considerare è la sequenza di $|V|$ vertici che costituiscono il ciclo hamiltoniano. L'algoritmo di verifica controlla che questa sequenza contenga ciascun vertice di V esattamente una volta e che con il primo vertice ripetuto alla fine formi un ciclo in G. Questo controllo può essere eseguito in tempo polinomiale.

Si dimostra adesso che CICLO-HAMILTONIANO è NP-completo facendo vedere che 3-SODDISFATTIBILITÀ-FNC $\leq_p$ CICLO-HAMILTONIANO. Data un formula booleana in 3-FNC ϕ sulle variabili $x_1, x_2, \ldots, x_n$ con clausole $C_1, C_2, \ldots, C_k$, ciascuna contenente esattamente 3 lettere distinte, si costruisce in tempo polinomiale un grafo $G = (V, E)$ che ha un ciclo hamiltoniano se e solo se ϕ è soddisfattibile. La costruzione è basata sulle ***sottostrutture***, che sono pezzi di grafi che garantiscono certe proprietà.

La prima sottostruttura è il sottografo A mostrato nella figura 36.15(a). Si supponga che A sia sottografo di un qualche grafo G e che le uniche connessioni tra A e la rimanente parte di G passino attraverso i vertici a, b, a' e b'. Si supponga inoltre che G abbia un ciclo hamiltoniano. Poiché ogni ciclo hamiltoniano dovrà passare dai vertici z_1, z_2, z_3 e z_4 in uno dei modi mostrati nella figura 36.15 (b) e (c), si può trattare il sottografo A come se fosse semplicemente composto da una coppia di archi (a, a') e (b, b') con il vincolo che qualsiasi

Figura 36.16 *La sottostruttura B, usata nella riduzione da* 3-SODDISFATTIBILITÀ-FNC *a* CICLO-HAMILTONIANO. *Nessun cammino dal vertice b_1 al vertice b_4 contenente tutti i vertici della sottostruttura può usare tutti e tre gli archi (b_1, b_2), (b_2, b_3) e (b_3, b_4). Tuttavia, può essere usato un qualsiasi sottoinsieme proprio di questi archi.* (a)-(e) *Cinque di questi sottoinsiemi.* (f) *Una rappresentazione di questa sottostruttura in cui almeno uno dei cammini indicati dalle frecce deve essere considerato in un ciclo hamiltoniano.*

ciclo hamiltoniano di G deve includere uno e solo uno di questi due archi. Si rappresenterà la sottostruttura A come mostrato nella figura 36.15(d).

Il sottografo B nella figura 36.16 è la seconda sottostruttura che ci interessa. Si supponga che B sia un sottografo di un qualche grafo G e che le uniche connessioni tra B ed il resto di G siano attraverso i vertici b_1, b_2, b_3 e b_4. Un ciclo hamiltoniano del grafo G non può attraversare *tutti* gli archi (b_1, b_2), (b_2, b_3) e (b_3, b_4), poiché altrimenti tutti i vertici della

sottostruttura diversi da b_1, b_2, b_3 e b_4 verrebbero saltati. Un ciclo hamiltoniano di g può, tuttavia, attraversare un sottoinsieme proprio di questi archi. Nelle Figure 36.16 (a)-(e) si mostrano cinque di questi sottoinsiemi; i rimanenti due sottoinsiemi possono essere ottenuti capovolgendo le parti (b) ed (e) dall'alto al basso. Questa sottostruttura si rappresenta come nella figura 36.16(f), l'idea di base è che in un ciclo hamiltoniano deve essere preso almeno uno dei cammini puntati con le frecce.

Il grafo G che si costruirà consiste principalmente di copie di queste due sottostrutture. La costruzione è illustrata nella figura 36.17. Per ciascuna delle k clausole di ϕ, si include una copia della sottostruttura B, e queste sottostrutture si uniscono tra loro come segue. Ponendo che b_{ij} sia la copia del vertice b_j nella i-esima copia della sottostruttura B, si connette $b_{i,4}$ a $b_{i+1,1}$ per $i = 1, 2, \ldots, k-1$.

Poi, per ciascuna variabile x_m di ϕ, si includono due vertici x'_m e x''_m. Si connettono questi due vertici attraverso due copie dell'arco (x'_m, x''_m), che si denotano con e_m e $\bar{e}_m$. L'idea è che se il ciclo hamiltoniano prende l'arco e_m, allora è come se alla variabile x_m fosse assegnato 1. Se il ciclo hamiltoniano prende l'arco $\bar{e}_m$, allora alla variabile è assegnato il valore 0. Ciascuna coppia di questi archi forma un ciclo di due archi; si connettono in serie questi piccoli cicli aggiungendo gli archi (x''_m, x'_{m+1}) per $m = 1, 2, \ldots, n-1$. Si connette la parte sinistra del grafo (le clausole) con la parte destra (le variabili) attraverso i due archi $(b_{1,1}, x'_1)$ e $(b_{k,4}, x''_n)$, che sono rispettivamente l'arco più in alto e quello più in basso nella figura 36.17.

Non abbiamo ancora finito con la costruzione del grafo G, poiché si devono ancora mettere in relazione le clausole con le variabili. Se la j-esima lettera della clausola C_i è x_m, allora si usa una sottostruttura di tipo A per connettere l'arco $(b_{ij}, b_{i,j+1})$ con l'arco e_m. Se la j-esima lettera della clausola C_i è $\neg x_m$, invece, si utilizza una sottostruttura A tra $(b_{ij}, b_{i,j+1})$ e $\bar{e}_m$. Per esempio, nella figura 36.17, dato che la clausola C_2 è $(x_1 \vee \neg x_2 \vee x_3)$, si dispongono tre sottostrutture A come segue:

- tra $(b_{2,1}, b_{2,2})$ e e_1,
- tra $(b_{2,2}, b_{2,3})$ e $\bar{e}_2$,
- tra $(b_{2,3}, b_{2,4})$ e e_3.

Si noti che la connessione di due archi attraverso una sottostruttura A in realtà comporta la sostituzione di ciascun arco con i cinque archi in alto o in basso della figura 36.15(a) e, naturalmente, anche l'aggiunta delle connessioni che passano attraverso i vertici z. Una data lettera l_m può apparire in diverse clausole (per esempio $\neg x_3$ nella figura 36.17), e quindi un arco e_m o $\bar{e}_m$ può essere interessato da diverse sottostrutture A (per esempio, l'arco $\bar{e}_3$). In questo caso le sottostrutture A si connettono in serie, come mostrato nella figura 36.18, sostituendo l'arco e_m o $\bar{e}_m$ con una serie di archi.

La formula ϕ è soddisfattibile se e solo se il grafo G contiene un ciclo hamiltoniano. Si supponga che G abbia un ciclo hamiltoniano h: allora si mostra che ϕ è soddisfattibile. Il ciclo h deve avere una forma particolare:

- Innanzitutto, attraversa l'arco $(b_{1,1}, x'_1)$ per andare da sinistra in alto a destra in alto.
- Poi segue tutti i vertici x'_m e x''_m dall'alto verso il basso, scegliendo l'arco e_m oppure l'arco $\bar{e}_m$, ma non entrambi.
- Quindi attraversa l'arco $(b_{k,4}, x''_n)$ per tornare indietro nella parte sinistra.

Figura 36.17 *Il grafo G costruito dalla formula* $\phi = (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$. *Un assegnamento soddisfacente s per le variabili di* ϕ *è* $s(x_1) = 0$, $s(x_2) = 1$, $s(x_3) = 1$, *che corrisponde al ciclo hamiltoniano mostrato. Si noti che se* $s(x_m) = 1$ *allora l'arco* e_m *è nel ciclo hamiltoniano, mentre se* $s(x_m) = 0$, *allora nel ciclo hamiltoniano vi è l'arco* $\bar{e}_m$.

- Infine attraversa le sottostrutture B dal basso all'alto sulla sinistra.

(In effetti esso attraversa anche gli archi dentro le sottostrutture A, ma questi sottografi sono usati soltanto per garantire che il cammino passi da una parte o dall'altra.)

Dato il ciclo hamiltoniano h, si definisce un assegnamento di verità per ϕ come segue. Se l'arco e_m appartiene ad h, allora si ha $x_m = 1$. Altrimenti, l'arco $\bar{e}_m$ appartiene ad h, e quindi si ha $x_m = 0$.

Quest'assegnamento soddisfa ϕ. Si consideri infatti una clausola C_i e la corrispondente sottostruttura B in G. Ciascun arco $(b_{i,j}, b_{i,j+1})$ è connesso tramite una sottostruttura A all'arco e_m o all'arco $\bar{e}_m$ a seconda che x_m o rispettivamente $\neg x_m$ sia la j-esima lettera nella clausola. L'arco $(b_{i,j}, b_{i,j+1})$ è attraversato da h se e solo se la corrispondente lettera è zero. Poiché ciascuno dei tre archi $(b_{i,1}, b_{i,2})$, $(b_{i,2}, b_{i,3})$, $(b_{i,3}, b_{i,4})$ della clausola C_i è anche in una sottostruttura B, non possono essere attraversati tutti e tre insieme dal ciclo hamiltoniano h.

Figura 36.18 *La costruzione effettivamente usata quando un arco* e_m *o* $\bar{e}_m$ *è influenzato da più sottostrutture di tipo A.* (a) *Una porzione della figura 36.17.* (b) *Il sottografo effettivamente costruito.*

Perciò uno di questi tre archi deve avere una corrispondente lettera a cui è assegnato il valore 1, e quindi la clausola C è soddisfatta. Questa proprietà vale per ciascuna clausola C_i, $i = 1, 2, \ldots, k$, e quindi la formula ϕ è soddisfatta.

Viceversa, si supponga che la formula ϕ sia soddisfatta per un qualche assegnamento di verità. Seguendo le regole precedenti, si può costruire un ciclo hamiltoniano per il grafo G: esso attraversa l'arco e_m se $x_m = 1$, attraversa l'arco $\bar{e}_m$ se $x_m = 0$, e attraversa l'arco $(b_{i,j}, b_{i,j+1})$ se e solo se la j-esima lettera della clausola C_i è zero per l'assegnamento considerato. Queste regole possono essere sicuramente applicate poiché si è assunto che s sia un assegnamento soddisfacente la formula ϕ.

In conclusione, facciamo notare che il grafo G può essere costruito in tempo polinomiale. Esso contiene una sottostruttura B per ciascuna delle k clausole in ϕ. C'è una sottostruttura A per ogni istanza di ciascuna lettera in ϕ, per cui ci sono $3k$ sottostrutture A. Poiché le sottostrutture A e B hanno una dimensione fissata, il grafo G ha $O(k)$ vertici e archi e si costruisce facilmente in tempo polinomiale. Quindi si è fornita una riduzione in tempo polinomiale da 3-SODDISFATTIBILITÀ-FNC a CICLO-HAMILTONIANO. ■

36.5.5 Il problema del commesso viaggiatore

Nel ***problema del commesso viaggiatore***[4], che è fortemente collegato al problema del ciclo hamiltoniano, un piazzista deve visitare n città. Modellando il problema come un grafo completo con n vertici, si può dire che il piazzista desidera fare un ***giro***, o ciclo hamiltoniano,

[4] N.d.R. In inglese, *Traveling-Salesman Problem*, abbreviato in TSP.

Figura 36.19 *Un'istanza del problema del commesso viaggiatore. Gli archi evidenziati in grigio rappresentano un giro di costo minimo, con costo 7.*

visitando ogni città esattamente una volta e terminando nella città da cui ha cominciato. È previsto un costo intero $c(i, j)$ per viaggiare dalla città i alla città j, ed il piazzista desidera fare il giro il cui costo totale sia minimo, dove il costo totale è la somma dei singoli costi lungo gli archi del giro. Per esempio, nella figura 36.19, un giro con costo minimo è $\langle u, w, v, x, u \rangle$, con costo 7. Il linguaggio formale per il corrispondente problema di decisione è

$$\begin{aligned} \text{TSP} = \{\langle G, c, k\rangle : \quad & G = (V, E) \text{ è un grafo completo,} \\ & c \text{ è una funzione da } V \times V \to \mathbf{Z}, \\ & k \in \mathbf{Z} \text{ e} \\ & G \text{ ha un giro per il commesso viaggiatore di costo al più } k\}. \end{aligned}$$

Il seguente teorema mostra che è inverosimile che esista un algoritmo veloce per il problema del commesso viaggiatore.

Teorema 36.15

Il problema del commesso viaggiatore è NP-completo.

Dimostrazione. Si mostra prima che TSP appartiene ad NP. Data un'istanza del problema si usa come certificato la sequenza degli n vertici del giro. L'algoritmo di verifica controlla che questa sequenza contenga ogni vertice esattamente una volta, somma i costi degli archi e controlla che la somma sia al più k. Questo processo può sicuramente essere svolto in tempo polinomiale.

Per dimostrare che TSP è NP-arduo, si mostra che CICLO-HAMILTONIANO $\leq_p$ TSP. Sia $G = (V, E)$ un'istanza di CICLO-HAMILTONIANO. Si costruisce un'istanza di TSP come segue. Si forma il grafo completo $G' = (V, E')$ dove $E' = \{(i, j) : i, j \in V\}$ e si definisce la funzione costo c come

$$c(i, j) = \begin{cases} 0 & \text{se } (i, j) \in E, \\ 1 & \text{se } (i, j) \notin E. \end{cases}$$

L'istanza di TSP è quindi $(G', c, 0)$ e viene facilmente determinata in tempo polinomiale.

Si mostra ora che il grafo G ha un ciclo hamiltoniano se e solo se il grafo G' ha un giro di costo al più 0. Si supponga che il grafo G abbia un ciclo hamiltoniano h. Ciascun arco in h appartiene ad E e quindi ha costo 0 in G'. Quindi, h è un giro di G' con costo 0. Viceversa, si supponga che il grafo G' abbia un giro h' di costo al più 0. Poiché il costo degli archi in E' è 0 o 1, il costo del giro h' è esattamente 0. Perciò h' contiene solo archi di E. Si conclude che h' è un ciclo hamiltoniano del grafo G. ■

Esercizi

36.5-1 Il ***problema del sottografo isomorfo***, dati due grafi G_1 e G_2, richiede se G_1 è isomorfo a un sottografo di G_2. Si mostri che il problema del sottografo isomorfo è NP-completo.

36.5-2 Data una matrice A $m \times n$ di interi e un vettore b di m interi, il ***problema della programmazione intera 0-1*** richiede se esiste un vettore x di n interi con elementi nell'insieme $\{0, 1\}$ tale che $Ax \leq b$. Si dimostri che il problema della programmazione intera 0 - 1 è NP-completo. (*Suggerimento:* lo si riduca da 3-SODDISFATTIBILITÀ-FNC.)

36.5-3 Si mostri che il problema della somma di sottoinsieme è risolvibile in tempo polinomiale se il valore obiettivo t è espresso in unario.

36.5-4 Il ***problema della partizione di un insieme*** prende come input un insieme S di numeri. Il quesito è se i numeri possono essere partizionati in due insiemi A e $\bar{A} = S - A$ tali che $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$. Si mostri che il problema della partizione di un insieme è NP-completo.

36.5-5 Si mostri che il problema del cammino hamiltoniano è NP-completo.

36.5-6 Il ***problema del ciclo semplice più lungo*** è il problema di determinare un ciclo semplice (senza vertici ripetuti) di lunghezza massima in un grafo. Si mostri che questo problema è NP-completo.

36.5-7 Il professor Marconi sostiene che il sottografo usato come sottostruttura A nella dimostrazione del Teorema 36.14 è più complicato del necessario: i vertici z_3 e z_4 della figura 36.15(a) ed i vertici che si trovano sopra e sotto ad essi non sono necessari. Il professore ha ragione? Cioè, la riduzione funziona anche con questa versione più piccola della sottostruttura, oppure la proprietà che il cammino passi da una parte o dall'altra della sottostruttura non è più valida?

Problemi

36-1 ***Insieme indipendente***

Un ***insieme indipendente*** di un grafo $G = (V, E)$ è un sottoinsieme $V' \subseteq V$ di vertici tale che ciascun arco di E è incidente in al più un vertice di V'. Il ***problema dell'insieme indipendente*** è di trovare in G un insieme indipendente di dimensione massima.

a. Si formuli il problema di decisione corrispondente al problema dell'insieme indipendente, e si dimostri che è NP-completo. (*Suggerimento:* lo si riduca dal problema della cricca.)

b. Si supponga di avere a disposizione un sottoprogramma per risolvere il problema di decisione definito nella parte (a). Si fornisca un algoritmo per trovare l'insieme indipendente di dimensione massima. Il tempo di esecuzione di questo algoritmo dovrebbe essere polinomiale rispetto a $|V|$ e $|E|$, quando ogni chiamata al sottoprogramma è contata come

un passo singolo.

Benché il problema di decisione dell'insieme indipendente sia NP-completo, certi casi speciali sono risolvibili in tempo polinomiale.

c. Si fornisca un algoritmo efficiente per risolvere il problema dell'insieme indipendente quando ogni vertice di G ha grado 2. Si analizzi il suo tempo di esecuzione, e si dimostri che l'algoritmo è corretto.

d. Si fornisca un algoritmo efficiente per risolvere il problema dell'insieme indipendente quando G è bipartito. Si analizzi il suo tempo di esecuzione, e si dimostri che l'algoritmo è corretto. (*Suggerimento*: si usino i risultati del paragrafo 27.3.)

36-2 *La colorazione di un grafo*

Una ***k-colorazione*** di un grafo non orientato $G = (V, E)$ è una funzione $c : V \rightarrow \{1, 2, \ldots, k\}$ tale che $c(u) \neq c(v)$ per ogni arco $(u, v) \in E$. In altre parole, i numeri $1, 2, \ldots, k$ rappresentano k colori, e vertici adiacenti devono avere colori diversi. Il ***problema della colorazione di un grafo*** è di determinare il minimo numero di colori necessari a colorare un dato grafo.

a. Si fornisca un algoritmo efficiente per determinare, se esiste, una 2-colorazione di un grafo.

b. Si riformuli il problema della colorazione di un grafo come problema di decisione. Si mostri che tale problema di decisione è risolvibile in tempo polinomiale se e solo se il problema della colorazione di un grafo è risolvibile in tempo polinomiale.

c. Sia 3-COLORI il linguaggio dei grafi che possono avere una 3-colorazione. Si mostri che se 3-COLORI è NP-completo, allora il problema di decisione della parte (b) è NP-completo.

Per dimostrare che 3-COLORI è NP-completo, si può usare una riduzione da 3-SODDISFATTIBILITÀ-FNC. Data una formula ϕ di m clausole in n variabili $x_1, x_2, \ldots, x_n$, si costruisce un grafo $G = (V, E)$ come segue. L'insieme V consiste di un vertice per ogni variabile, un vertice per la negazione della variabile, 5 vertici per ogni clausola e 3 vertici speciali: TRUE, FALSE, e RED. Gli archi del grafo sono di due tipi: archi "lettere" che sono indipendenti dalle clausole, e archi "clausole" che dipendono dalle clausole. Gli archi lettere formano un triangolo sui vertici speciali e formano un triangolo anche per x_i, $\neg x_i$ e RED, per $i = 1, 2, \ldots, n$.

d. Si mostri che in qualsiasi 3-colorazione c di un grafo contenente gli archi lettere, quando la variabile o la sua negazione è colorata c(TRUE), l'altra (la negazione o la variabile) è colorata c(FALSE). Si deduca che per qualsiasi assegnamento di verità per ϕ, c'è una 3-colorazione del grafo che contiene proprio gli archi lettere.

La sottostruttura mostrata nella figura 36.20 è usata per forzare la condizione corrispondente alla clausola $(x \vee y \vee z)$. Ciascuna clausola richiede un'unica copia dei 5 vertici che nella figura sono colorati di nero. Essi sono connessi alle lettere della clausola e al vertice speciale TRUE come mostrato.

e. Si mostri che se ciascuno tra x, y, e z è colorato c(TRUE) o c(FALSE), allora la sottostruttura

Figura 36.20 *La sottostruttura corrispondente alla clausola* $(x \vee y \vee z)$, *usata nel Problema* 36.2.

è 3-colorabile se e solo se almeno uno tra x, y, e z è colorato c(TRUE).

f. Si completi la dimostrazione che 3-COLORI è NP-completo.

Note al capitolo

Garey e Johnson [79] forniscono una magnifica guida alla NP-completezza, discutendo a fondo la teoria e fornendo un catalogo di molti problemi NP-completi che erano noti nel 1979. (L'elenco delle aree dei problemi NP-completi che si trova all'inizio del paragrafo 36.5 è preso dal loro indice.) Hopcroft e Ullman [104] e Lewis e Papadimitriou [139] presentano una buona trattazione della NP-completezza nel contesto della teoria della complessità. Anche in Aho, Hopcroft e Ullman [4] si esamina la NP-completezza e si forniscono diverse riduzioni, inclusa una per il problema della copertura di vertici a partire dal problema del ciclo hamiltoniano.

La classe P fu introdotta nel 1964 da Cobham[44] e, indipendentemente, nel 1965 da Edmonds [61], che introdusse anche la classe NP e congetturò che P ≠ NP. La definizione di NP-completezza fu proposta da Cook [49] nel 1971, che fornì le prime dimostrazioni di NP-completezza per i problemi della soddisfattibilità di formule e della soddisfattibilità di formule 3-FNC. Indipendentemente, Lèvin [138] arrivò alla stessa nozione, fornendo una dimostrazione di NP-completezza per un problema di tassellatura. Nel 1972, Karp [116] introdusse la metodologia delle riduzioni e dimostrò una ricca varietà di problemi NP-completi. La pubblicazione di Karp include le dimostrazioni originali di NP-completezza dei problemi di copertura di vertici, cricca e ciclo hamiltoniano. Da allora, centinaia di problemi sono stati dimostrati NP-completi da molti ricercatori.

La dimostrazione del Teorema 36.14 è basata su quella che si trova in Papadimitriou e Steiglitz [154].

Algoritmi approssimati

37

Molti problemi di interesse pratico sono NP-completi e risultano troppo importanti per poterli tralasciare semplicemente perché la loro soluzione ottima è intrattabile. Se un problema è NP-completo, è inverosimile che si possa trovare un algoritmo con tempo polinomiale che lo risolva esattamente, ma ciò non significa che siano perse tutte le speranze. Ci sono due approcci per aggirare la NP-completezza. In primo luogo, se gli input effettivi sono piccoli, allora un algoritmo con tempo di esecuzione esponenziale può essere perfettamente soddisfacente. In secondo luogo, può essere ancora possibile trovare soluzioni *quasi ottime* in tempo polinomiale (sia nel caso peggiore che nel caso medio). In pratica, una soluzione quasi ottima è spesso soddisfacente. Un algoritmo che restituisce soluzioni quasi ottime è chiamato un ***algoritmo approssimato***. Questo capitolo presenta algoritmi approssimati con tempo polinomiale relativi a diversi problemi NP-completi.

Limiti delle prestazioni di algoritmi approssimati

Si supponga di lavorare ad un problema di ottimizzazione in cui ogni potenziale soluzione abbia un costo positivo e si desideri trovare una soluzione quasi ottima. In funzione del problema, una soluzione ottima può essere definita come una soluzione con il massimo costo possibile o una con il minimo costo possibile: il problema può essere di massimo o di minimo.

Si dice che un algoritmo approssimato per il problema ha ***rapporto limite*** $\rho(n)$ se per ogni input di dimensione n, il costo C della soluzione prodotta dall'algoritmo approssimato si discosta al più per un fattore $\rho(n)$ dal costo C^* di una soluzione ottima:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n). \tag{37.1}$$

Questa definizione si applica a problemi sia di massimo sia di minimo. Per un problema di massimo, $0 < C \le C^*$ ed il rapporto C^*/C dà il fattore per cui una soluzione ottima è più grande di quella approssimata. Analogamente, per un problema di minimo, $0 < C^* \le C$ ed il rapporto C/C^* fornisce il fattore per il quale il costo della soluzione approssimata risulta essere più grande del costo di una soluzione ottima. Poiché si è assunto che tutte le soluzione abbiano costo positivo, questi rapporti sono sempre ben definiti. Il rapporto limite trovato con un algoritmo approssimato non è mai minore di 1, poiché $C/C^* < 1$ implica $C^*/C > 1$. Un algoritmo ottimo ha rapporto limite uguale ad 1, ed un algoritmo approssimato con rapporto limite grande può fornire una soluzione nettamente peggiore di quella ottima.

Talvolta è più conveniente operare con una misura dell'errore relativo. Per qualsiasi input, l'***errore relativo*** di un algoritmo approssimato è definito essere

$$\frac{|C - C^*|}{C^*},$$

dove, come prima, C^* è il costo di una soluzione ottima e C è il costo della soluzione prodotta dall'algoritmo approssimato. L'errore relativo è sempre non negativo. Un algoritmo approssimato ha un ***errore relativo limitato da*** $\varepsilon(n)$ se

$$\frac{|C - C^*|}{C^*} \le \varepsilon(n). \tag{37.2}$$

Dalle definizioni segue che l'errore relativo può essere limitato da una funzione del rapporto limite

$$\varepsilon(n) \le \rho(n) - 1 . \tag{37.3}$$

(Per un problema di minimo questa risulta essere un'uguaglianza, mentre per un problema di massimo si ha $\varepsilon(n) = (\rho(n) - 1)/\rho(n)$, che soddisfa la disuguaglianza (37.3) dato che $\rho(n) \ge 1$.)

Per molti problemi, sono stati sviluppati algoritmi approssimati che hanno un rapporto limite fisso, indipendente da n. Per questi problemi si userà, per semplicità, la notazione ε o ρ per indicare la non dipendenza da n.

Per altri problemi, gli informatici non sono stati ancora in grado di determinare alcun algoritmo approssimato con tempo polinomiale con un rapporto limite fisso. Per tali problemi, la miglior cosa che si può fare è lasciare crescere il rapporto limite in funzione della dimensione n dell'input. Un esempio di tale problema è il problema della copertura di un insieme presentato nel paragrafo 37.3.

Per alcuni problemi NP-completi esistono algoritmi approssimati che possono raggiungere rapporti limite sempre più piccoli (o, equivalentemente, raggiungere errori relativi sempre più piccoli) usando sempre più tempo di calcolo. Cioè c'è una relazione tra il tempo di calcolo e la qualità dell'approssimazione. Un esempio è il problema della somma di sottoinsieme studiato nel paragrafo 37.4. Questo aspetto è tanto importante da meritare una definizione propria.

Uno ***schema di approssimazione*** per un problema di ottimizzazione è un algoritmo approssimato che prende come input non solo un'istanza del problema ma anche un valore $\varepsilon > 0$ tale che, per qualsiasi ε fissato, lo schema è un algoritmo approssimato con errore relativo limitato da ε. Si dice che uno schema di approssimazione è uno ***schema di approssimazione polinomiale*** se per qualsiasi $\varepsilon > 0$ fissato, lo schema impiega un tempo polinomiale rispetto alla dimensione n dell'input dell'istanza.

Il tempo di esecuzione di uno schema di approssimazione polinomiale non dovrebbe crescere troppo rapidamente al decrescere di ε. Idealmente, se ε diminuisce di un fattore costante, il tempo di esecuzione per raggiungere l'approssimazione voluta non dovrebbe aumentare per più di un fattore costante. In altre parole, si vuole che il tempo di esecuzione sia polinomiale rispetto a $1/\varepsilon$ così come lo è rispetto a n.

Si dice che lo schema è ***uno schema di approssimazione pienamente polinomiale*** se il suo tempo di esecuzione è polinomiale sia per $1/\varepsilon$ che per la dimensione n dell'input dell'istanza. Per esempio, lo schema potrebbe avere tempo di esecuzione $(1/\varepsilon)^2 n^3$. Con un tale schema, qualsiasi diminuzione di ε di un fattore costante può essere ottenuta con un corrispondente aumento del tempo di esecuzione di un fattore costante.

Descrizione del capitolo

I primi tre paragrafi di questo capitolo presentano alcuni esempi di algoritmi approssimati con tempo polinomiale per problemi NP-completi e l'ultimo paragrafo presenta uno schema di approssimazione pienamente polinomiale. Il paragrafo 37.1 comincia con lo studio del problema della copertura di vertici, un problema di minimizzazione NP-completo, con un algoritmo approssimato con rapporto limite fissato a 2. Il paragrafo 37.2 presenta un algoritmo approssimato con rapporto limite fissato a 2 per il problema del commesso viaggiatore, in cui la funzione di costo soddisfa la disuguaglianza triangolare. Esso mostra anche che senza la disuguaglianza triangolare un algoritmo approssimato con rapporto limite fissato non può esistere a meno che P = NP. Nel paragrafo 37.3 si mostra come un metodo greedy possa essere usato come un algoritmo approssimato efficiente per il problema della copertura di un insieme, ottenendo un copertura il cui costo è, nel peggiore dei casi, più grande del costo ottimo di un fattore logaritmico. In conclusione, il paragrafo 37.4 presenta uno schema di approssimazione pienamente polinomiale per il problema della somma di sottoinsieme.

37.1 Il problema della copertura di vertici

Il problema della copertura di vertici come problema di decisione è stato definito nel paragrafo 36.5.2, dove si è anche dimostrata la sua NP-completezza. Una ***copertura di vertici*** di un grafo non orientato $G = (V, E)$ è un sottoinsieme $V' \subseteq V$ tale che se (u, v) è un arco di G allora $u \in V'$ oppure $v \in V'$ (o entrambi). La dimensione di una copertura di vertici è il numero dei vertici che contiene.

Il ***problema della copertura di vertici*** è quello di trovare una copertura di vertici di dimensione minima, per un dato grafo non orientato. Una tale copertura di vertici si dice ***copertura ottima di vertici***. Per il Teorema 36.12, questo problema è NP-arduo poiché il relativo problema di decisione è NP-completo.

Anche se può essere difficoltoso trovare una copertura ottima di vertici di un grafo G, tuttavia non è così complicato trovare una copertura di vertici che sia quasi ottima. Il seguente algoritmo approssimato prende come input un grafo G non orientato e restituisce una copertura di vertici la cui dimensione è garantita essere al più doppia di quella di una copertura ottima di vertici.

```
APPROX-VERTEX-COVER(G)
C ← ∅
E' ← E[G]
while E' ≠ ∅
        do   sia (u, v) un generico arco di E'
             C ← C ∪ {u, v}
             si rimuova da E' ogni arco incidente in u o in v
return C
```

Figura 37.1 *Il funzionamento di* APPROX-VERTEX-COVER. (a) *Il grafo di input G, che ha 7 vertici e 8 archi.* (b) *L'arco (b, c), mostrato in grigio scuro, è il primo arco scelto da* APPROX-VERTEX-COVER. *I vertici b e c, mostrati in grigio chiaro, sono aggiunti all'insieme C contenente i vertici della copertura che si sta creando. Gli archi (a, b), (c, e) e (c, d), tratteggiati, sono rimossi dato che adesso sono coperti da alcuni vertici in C.* (c) *L'arco (e, f) è aggiunto a C.* (d) *L'arco (d, g) è aggiunto a C.* (e) *L'insieme C, che è la copertura di vertici prodotta da* APPROX-VERTEX-COVER, *contiene sei vertici b, c, d, e, f, g.* (f) *La copertura ottima di vertici per questo problema contiene solo tre vertici: b, d ed e.*

La figura 37.1 illustra il funzionamento di APPROX-VERTEX-COVER. La variabile C contiene la copertura di vertici che si sta costruendo. La linea 1 inizializza C all'insieme vuoto. La linea 2 assegna ad E' una copia dell'insieme $E[G]$ degli archi del grafo dato. Il ciclo delle linee 3-6 iterativamente toglie un arco (u, v) da E', aggiunge i suoi estremi a C e cancella tutti gli archi in E' che sono coperti da u o da v. Il tempo di esecuzione di questo algoritmo è $O(E)$, usando un'appropriata struttura di dati per rappresentare E'.

Teorema 37.1

APPROX-VERTEX-COVER ha rapporto limite uguale a 2.

Dimostrazione. L'insieme di vertici C restituito da APPROX-VERTEX-COVER è una copertura di vertici, dato che l'algoritmo cicla finché tutti gli archi in $E[G]$ siano coperti da qualche vertice in C.

Per verificare che APPROX-VERTEX-COVER restituisce una copertura di vertici che è al più doppia della dimensione di una copertura ottima, si denoti con A l'insieme degli archi selezionati nella linea 4 di APPROX-VERTEX-COVER. Nessuna coppia di archi di A condivide un estremo, dato che per ciascun arco selezionato nella linea 4, tutti gli altri archi incidenti nei suoi estremi sono cancellati da E' nella linea 6. Perciò ogni esecuzione della linea 6 aggiunge due nuovi vertici a C e $|C| = 2\,|A|$. Tuttavia, per coprire gli archi in A qualsiasi copertura di vertici – ed in particolare una copertura ottima C^* – deve includere almeno un estremo di

ciascun arco di A. Poiché nessuna coppia di archi in A condivide un estremo, nessun vertice della copertura è incidente su più di un arco di A. Perciò $|A| \le |C^*|$, e $|C| \le 2\,|C^*|$, cosicché il teorema risulta dimostrato. ■

Esercizi

37.1-1 Si fornisca l'esempio di un grafo per cui APPROX-VERTEX-COVER produce sempre una soluzione non ottima.

37.1-2 Il professor Nixon propone la seguente euristica per risolvere il problema della copertura di vertici. Si selezioni ricorsivamente un vertice con il grado più alto, rimuovendo poi tutti i suoi archi incidenti. Si fornisca un esempio che mostri che l'euristica del professore non ha rapporto limite uguale a 2.

37.1-3 Si fornisca un algoritmo greedy efficiente che trovi una copertura di vertici ottima di un albero, in un tempo lineare.

37.1-4 Con la dimostrazione del Teorema 36.12, si è ottenuto anche che il problema della copertura di vertici ed il problema della cricca, entrambi NP-completi, sono complementari nel senso che una copertura di vertici ottima è il complemento di una cricca di dimensione massima nel grafo complementare. Questa relazione implica che ci possa essere un algoritmo approssimato con rapporto limite costante per il problema della cricca? Si giustifichi la risposta.

37.2 Il problema del commesso viaggiatore

Nel problema del commesso viaggiatore introdotto nel paragrafo 36.5.5, dato un grafo completo non orientato $G = (V, E)$ con un costo intero non negativo $c(u, v)$ associato a ciascun arco $(u, v) \in E$, si deve trovare un ciclo hamiltoniano (un giro) di G che abbia costo minimo. Come estensione della notazione già vista, si indichi con $c(A)$ il costo complessivo di tutti gli archi del sottoinsieme $A \subseteq E$:

$$c(A) = \sum_{(u,v) \in A} c(u, v) .$$

In molte situazioni pratiche, risulta sempre più economico andare direttamente dal posto u al posto w: attraversare fermate intermedie v non può essere meno costoso. Detto con altre parole, il salto di una fermata non può mai aumentare il costo. Questa nozione si formalizza dicendo che la funzione di costo c soddisfa la ***disuguaglianza triangolare*** se per tutti i vertici $u, v, w \in V$

$$c(u, w) \le c(u, v) + c(v, w).$$

La disuguaglianza triangolare è abbastanza naturale ed in molte applicazioni risulta automaticamente soddisfatta. Per esempio, se i vertici di un grafo sono punti del piano ed il costo del trasferimento tra due vertici è la comune distanza euclidea tra essi, la disuguaglianza triangolare è soddisfatta.

Come è mostrato nell'Esercizio 37.2-1, assumere che la funzione di costo soddisfi la disuguaglianza triangolare non altera la NP-completezza del problema del commesso viaggiatore. Quindi è inverosimile che si riesca a trovare un algoritmo con tempo polinomiale per risolvere questo problema in modo esatto. Invece, si possono cercare buoni algoritmi approssimati.

Nel paragrafo 37.2.1 si esamina un algoritmo approssimato per il problema del commesso viaggiatore con la disuguaglianza triangolare che ha rapporto limite uguale a 2. Nel paragrafo 37.2.2 si mostra che, senza la disuguaglianza triangolare, un algoritmo approssimato con rapporto limite costante non esiste a meno che P = NP.

37.2.1 Il problema del commesso viaggiatore con la disuguaglianza triangolare

Il seguente algoritmo calcola un giro quasi ottimo su un grafo non orientato G, usando l'algoritmo MST-PRIM del paragrafo 24.2 per trovare il minimo albero di copertura. Si vedrà che quando la funzione di costo soddisfa la disuguaglianza triangolare, il giro che questo algoritmo fornisce non è lungo più del doppio della lunghezza di un giro ottimo.

```
APPROX-TSP-TOUR(G, c)
seleziona un vertice r ∈ V[G] come vertice "radice"
costruisci un minimo albero di copertura T per G dalla radice r usando MST-PRIM(G,c,r)
sia L la lista dei vertici visitati con la visita in ordine anticipato di T
return il ciclo hamiltoniano H che visita i vertici nell'ordine di L
```

Si ricorda, dal paragrafo 13.1, che una visita in ordine anticipato di un albero visita ricorsivamente tutti i vertici dell'albero, elencando un vertice non appena questo viene incontrato e prima che qualsiasi suo figlio venga visitato.

La figura 37.2 illustra il funzionamento di APPROX-TSP-TOUR. La parte (a) della figura mostra l'insieme dei vertici dati, e la parte (b) mostra il minimo albero di copertura T costruito da MST-PRIM a partire dal vertice radice a. La parte (c) mostra come i vertici di T siano visitati in ordine anticipato e la parte (d) visualizza il giro corrispondente che è restituito da APPROX-TSP-TOUR. La parte (e) visualizza un giro ottimo, che è più corto di circa il 23%.

Il tempo di esecuzione di APPROX-TSP-TOUR è $\Theta(E) = \Theta(V^2)$, dato che l'input è un grafo completo (si veda l'Esercizio 24.2-2). Si mostrerà adesso che se la funzione di costo di un'istanza del problema del commesso viaggiatore soddisfa la disuguaglianza triangolare, allora APPROX-TSP-TOUR fornisce un giro il cui costo non è più grande del doppio del costo di un giro ottimo.

Teorema 37.2

APPROX-TSP-TOUR è un algoritmo approssimato con rapporto limite uguale a 2 per il problema del commesso viaggiatore che soddisfa la disuguaglianza triangolare.

Dimostrazione. Sia H^* una soluzione ottima per l'insieme dato dei vertici. Un enunciato equivalente del teorema è che $c(H) \le 2c(H^*)$, dove H è il giro calcolato da APPROX-TSP-TOUR. Poiché un albero di copertura si ottiene cancellando un arco qualsiasi da un giro, se T è un minimo albero di copertura per l'insieme dei vertici dati, allora

Figura 37.2 *Il funzionamento di* APPROX-TSP-TOUR. (a) *L'insieme dato dei punti sta sui vertici di un griglia di interi. Per esempio, f è una unità a destra e due unità sopra rispetto ad h. È usata come funzione di costo l'usuale distanza euclidea tra due punti.* (b) *Un minimo albero di copertura T di questi punti, come è calcolato da* MST-PRIM. *Il vertice a è il vertice radice. I vertici sono etichettati in modo tale da essere aggiunti all'albero principale da* MST-PRIM *in ordine alfabetico.* (c) *Una visita di T, cominciando da a. Una visita completa dell'albero raggiunge i nodi nel seguente ordine: a, b, c, b, h, b, a, d, e, f, e, g, e, d, a. Una visita in ordine anticipato di T elenca un vertice soltanto quando viene incontrato per la prima volta, producendo l'ordine a, b, c, h, d, e, f, g.* (d) *Un giro dei vertici ottenuto visitando i vertici nell'ordine dato dalla visita anticipata. Questo è il giro H restituito da* APPROX-TSP-TOUR. *Il suo costo totale è circa* 19.074. (e) *Un giro ottimo H^* per l'insieme dei vertici dati. Il suo costo totale è circa* 14.715.

$$c(T) \le c(H^*). \tag{37.4}$$

Una ***visita completa*** di T elenca i vertici quando sono visitati la prima volta e anche tutte le volte che si ritorna su di loro dopo una visita ad un sottoalbero. Si chiami W questa visita. La visita completa dell'esempio dato è

$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$.

Dato che la visita completa attraversa ogni arco di T esattamente due volte, si ha

$$c(W) = 2\,c(T). \tag{37.5}$$

Le equazioni (37.4) e (37.5) implicano che

$$c(W) \le 2\,c(H^*), \tag{37.6}$$

e quindi il costo di W non supera il doppio del costo di un giro ottimo.

Sfortunatamente, W non è generalmente un giro, poiché visita alcuni vertici più di una volta. Per la disuguaglianza triangolare, tuttavia, si può cancellare la visita di qualsiasi vertice di W senza che il costo aumenti. (Se un vertice v è cancellato da W tra le visite di u e w, il risultante ordinamento indica che si arriva a w direttamente da u.) Ripetendo l'applicazione di questa

operazione si possono rimuovere da W tutte le visite successive dello stesso vertice, tranne la prima. Nell'esempio, l'ordinamento risulta

a, b, c, h, d, e, f, g .

Questo ordinamento è uguale a quello ottenuto visitando l'albero T in ordine anticipato. Sia H il ciclo corrispondente a questa visita in ordine anticipato. È un ciclo hamiltoniano, poiché ogni vertice è visitato esattamente una volta, ed in realtà è il ciclo calcolato da APPROX-TSP-TOUR. Poiché H è ottenuto cancellando vertici dalla visita completa W, si ha

$$c(H) \le c(W). \tag{37.7}$$

Combinando le diseguaglianze (37.6) e (37.7) si completa la dimostrazione. ■

Malgrado il rapporto limite favorevole stabilito dal Teorema 37.2, generalmente APPROX-TSP-TOUR non è la miglior scelta pratica per questo problema. Ci sono altri algoritmi approssimati che in pratica hanno prestazioni migliori (si vedano i riferimenti alla fine di questo capitolo).

37.2.2 Il problema generale del commesso viaggiatore

Se si elimina l'ipotesi che la funzione di costo c soddisfi la disuguaglianza triangolare, non si possono trovare in tempo polinomiale buoni giri approssimati a meno che P = NP.

Teorema 37.3

Se P ≠ NP e $\rho \ge 1$, non c'è alcun algoritmo approssimato con tempo polinomiale con rapporto limite ρ per il problema generale del commesso viaggiatore.

Dimostrazione. La dimostrazione è per assurdo. Si supponga per assurdo che, per qualche numero $\rho \ge 1$, ci sia un algoritmo approssimato A con tempo polinomiale e rapporto limite ρ. Senza perdere di generalità si può supporre ρ intero, arrotondandolo per eccesso se necessario. Si mostrerà adesso come usare l'algoritmo A per risolvere istanze del problema del ciclo hamiltoniano (definito nel paragrafo 36.5.5) in tempo polinomiale. Poiché, per il Teorema 36.14, il problema del ciclo hamiltoniano è NP-completo risolverlo in tempo polinomiale implica che P = NP, per il Teorema 36.4.

Sia $G = (V, E)$ un'istanza del problema del ciclo hamiltoniano. Si desidera determinare in modo efficiente se G contiene un ciclo hamiltoniano facendo uso dell'algoritmo approssimato A ipotizzato. Si trasforma G in un'istanza del problema del commesso viaggiatore come segue. Sia $G' = (V, E')$ il grafo completo su V; cioè

$$E' = \{(u, v) : u, v \in V \text{ e } u \ne v\}.$$

Si assegni un costo intero a ciascun arco in E' come segue:

$$c(u, v) = \begin{cases} 1 & \text{se } (u, v) \in E, \\ \rho|V| + 1 & \text{altrimenti.} \end{cases}$$

Le rappresentazioni di G' e c possono essere create da una rappresentazione di G in tempo polinomiale rispetto a $|V|$ e $|E|$.

Si consideri ora il problema del commesso viaggiatore (G', c). Se il grafo originale G ha un ciclo hamiltoniano H, allora la funzione di costo c assegna a ciascun arco di H costo 1, per cui (G', c) contiene un giro di costo $|V|$. In altre parole, se G non contiene alcun ciclo Hamiltoniano, allora qualsiasi giro di G' deve usare qualche arco che non è in E.

Ma qualsiasi giro che usa un arco non in E ha un costo di almeno

$$(\rho|V| + 1) + (|V| - 1) > \rho|V|.$$

Dato che gli archi non in G sono così costosi, c'è una grande differenza tra il costo di un giro che sia un ciclo hamiltoniano in G (costo $|V|$) ed il costo di qualsiasi altro giro (costo maggiore di $\rho|V|$).

Cosa accade se si applica l'algoritmo di approssimazione A al problema del commesso viaggiatore (G', c)? Dato che è garantito che A restituisce un giro di costo non superiore a ρ volte il costo di un giro ottimo, se G contiene un ciclo hamiltoniano allora A lo deve restituire. Se G non ha un ciclo hamiltoniano, allora A restituisce un giro di costo maggiore di $\rho|V|$. Perciò, si può usare l'algoritmo A per risolvere il problema del ciclo hamiltoniano in tempo polinomiale. ■

Esercizi

37.2-1 Si mostri come si possa trasformare in tempo polinomiale un'istanza del problema del commesso viaggiatore in un'altra istanza la cui funzione di costo soddisfi la disuguaglianza triangolare. Le due istanze devono avere lo stesso insieme di soluzioni ottime. Si spieghi perché una tale trasformazione con tempo polinomiale non contraddica il Teorema 37.3, assumendo che P ≠ NP.

37.2-2 Si consideri la seguente ***euristica del punto più vicino*** per costruire l'approssimazione di un giro del commesso viaggiatore. Si cominci con un ciclo banale costituito da un singolo vertice scelto arbitrariamente. In ciascun passo, si identifica il vertice u che non è nel ciclo ma la cui distanza da qualsiasi vertice del ciclo è minima. Si supponga che il vertice del ciclo più vicino a u sia v. Si estende il ciclo per includere u inserendolo immediatamente dopo v. Si ripeta finché tutti i vertici sono nel ciclo. Si dimostri che questa euristica restituisce un giro il cui costo totale non è più del doppio del costo di un giro ottimo.

37.2-3 Il ***problema del commesso viaggiatore con collo di bottiglia*** è il problema di trovare il ciclo hamiltoniano tale che la lunghezza del più lungo arco del ciclo sia minima. Assumendo che la funzione di costo soddisfi la disuguaglianza triangolare, mostrare che per questo problema esiste un algoritmo approssimato con tempo polinomiale che ha rapporto limite 3. (*Suggerimento*: si mostri ricorsivamente che si possono visitare tutti i nodi di un minimo albero di copertura esattamente una volta, prendendo una visita completa dell'albero ed eliminando da essa i nodi in modo che non siano eliminati più di due nodi intermedi consecutivi. Si confronti anche l'Esercizio 24.2-6).

37.2-4 Si supponga che i vertici di un'istanza del problema del commesso viaggiatore siano punti del piano e che il costo $c(u, v)$ sia la distanza euclidea tra i punti u e v. Si mostri che un giro ottimo non attraversa mai se stesso.

37.3 Il problema della copertura di un insieme

Il problema della copertura di un insieme è un problema di ottimizzazione che modella molti problemi di allocazione di risorse. Esso generalizza il problema NP-completo della copertura di vertici e perciò è NP-arduo. L'algoritmo approssimato sviluppato per gestire il problema della copertura di vertici non può tuttavia essere applicato per questo problema, è quindi necessario tentare altri approcci. Si esaminerà una semplice euristica greedy che ha rapporto limite logaritmico. Cioè, al crescere della dimensione dell'istanza del problema, la dimensione della soluzione approssimata può crescere rispetto alla dimensione di una soluzione ottima. Tuttavia, dato che la funzione logaritmica cresce piuttosto lentamente, quest'algoritmo approssimato può non ostante ciò dare risultati utili.

Un'istanza $(X, \mathcal{F})$ del ***problema della copertura di un insieme*** consiste di un insieme finito X ed una famiglia $\mathcal{F}$ di sottoinsiemi di X, tale che ogni elemento di X appartiene ad almeno un sottoinsieme di $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S .$$

Si dice che un sottoinsieme $S \in \mathcal{F}$ è una ***copertura*** dei suoi elementi. Il problema è di trovare un sottoinsieme di dimensione minima $\mathcal{C} \subseteq \mathcal{F}$ i cui elementi siano una copertura di tutti quelli di X:

$$X = \bigcup_{S \in \mathcal{C}} S . \tag{37.8}$$

Qualsiasi $\mathcal{C}$ che soddisfi l'equazione (37.8) è detto una ***copertura*** di X. La figura 37.3 illustra questo problema.

Il problema della copertura di un insieme è l'astrazione di molti problemi comuni di tipo combinatorio. Per esempio, si supponga che X rappresenti un insieme di capacità necessarie a risolvere un problema e che ci sia un insieme di persone disponibili a lavorare sul problema. Si desidera formare un gruppo, contenente il minor numero possibile di persone, tale che per ogni capacità richiesta in X, ci sia un membro del gruppo che ha quella capacità. Nella versione decisionale del problema della copertura di un insieme, ci si chiede se esiste una copertura con dimensione al più k, dove k è un ulteriore parametro specificato nell'istanza del problema. La versione decisionale del problema è NP-completa, come è richiesto di mostrare nell'Esercizio 37.3-2.

Un algoritmo greedy approssimato

Il metodo greedy opera prendendo, in ciascun passo, l'insieme S che è copertura del maggior numero di elementi rimasti ancora non coperti.

Figura 37.3 *Un'istanza $(X, \mathcal{F})$ del problema della copertura di un insieme, dove X consiste dei 12 punti neri ed $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. Una copertura con dimensione minima è $\mathcal{C} = \{S_3, S_4, S_5\}$. L'algoritmo greedy approssimato produce una copertura di dimensione 4 selezionando, nell'ordine, gli insiemi S_1, S_4, S_6 e S_3.*

```
GREEDY-SET-COVER(X, F)
1  U ← X
2  C ← ∅
3  while U ≠ ∅
4      do seleziona un S ∈ F che massimizzi |S ∩ U|
5          U ← U − S
6          C ← C ∪ {S}
7  return C
```

Nell'esempio di figura 37.3, GREEDY-SET-COVER aggiunge a $\mathcal{C}$ gli insiemi S_1, S_4, S_5, S_3 nell'ordine.

L'algoritmo funziona come segue. L'insieme U contiene ad ogni passo l'insieme degli elementi che ancora non hanno una copertura. L'insieme $\mathcal{C}$ rappresenta la copertura che si sta costruendo. La linea 4 contiene il passo di decisione greedy: è scelto un sottoinsieme S che sia copertura del massimo numero di elementi ancora non coperti (con parità risolta arbitrariamente). Dopo che S è stato selezionato, i suoi elementi sono rimossi da U, ed S è posto in $\mathcal{C}$. Quando l'algoritmo termina, l'insieme $\mathcal{C}$ contiene una sottofamiglia di $\mathcal{F}$ che è copertura di X.

L'algoritmo GREEDY-SET-COVER può essere realizzato facilmente in modo da essere eseguito con tempo polinomiale rispetto a $|X|$ e $|\mathcal{F}|$. Poiché il numero delle iterazioni del ciclo delle linee 3-6 è al più $\min(|X|, |\mathcal{F}|)$ ed il corpo del ciclo può essere realizzato in modo da essere eseguito in tempo $O(|X|\,|\mathcal{F}|)$, allora esiste una realizzazione che richiede tempo $O(|X|\,|\mathcal{F}|\,\min(|X|, |\mathcal{F}|))$. L'Esercizio 37.3-3 richiede la definizione di un algoritmo con tempo lineare.

Analisi

Si mostra adesso che l'algoritmo greedy restituisce una copertura dell'insieme che non è troppo più grande di una copertura ottima. Per semplicità, in questo capitolo si denota il d-esimo numero armonico $H_d = \sum_{i=1}^{d} 1/i$ (si veda il paragrafo 3.1) con $H(d)$.

Teorema 37.4

GREEDY-SET-COVER ha rapporto limite

$H(\max\{|S| : S \in \mathcal{F}\})$

Dimostrazione. La dimostrazione procede assegnando un costo a ciascun insieme selezionato dall'algoritmo, distribuendo questo costo sugli elementi coperti per la prima volta, e quindi usando questo costo per derivare la relazione richiesta tra la dimensione di una copertura ottima dell'insieme, C^*, e la dimensione della copertura dell'insieme C restituita dall'algoritmo. Si denoti con S_i l'i-esimo sottoinsieme selezionato da GREEDY-SET-COVER; l'algoritmo richiede costo 1 quando aggiunge S_i a C. Si distribuisce questo costo di selezione di S_i uniformemente tra gli elementi coperti per la prima volta da S_i. Si denoti con c_x il costo assegnato all'elemento x, per ogni $x \in X$. A ciascun elemento è assegnato un costo solo una volta, quando è coperto per la prima volta. Se x è coperto per la prima volta da S_i, allora

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

L'algoritmo trova una soluzione C di costo totale $|C|$, e questo costo è suddiviso su tutti gli elementi di X. Perciò, dato che anche la soluzione ottima C^* è una copertura di X, si ha

$$\begin{aligned} |C| &= \sum_{x \in X} c_x \\ &\le \sum_{S \in C^*} \sum_{x \in S} c_x . \end{aligned} \tag{37.9}$$

Il resto della dimostrazione è basato sulla seguente disuguaglianza chiave, che sarà provata fra poco. Per qualsiasi insieme S appartenente alla famiglia $\mathcal{F}$,

$$\sum_{x \in S} c_x \le H(|S|) . \tag{37.10}$$

Dalle disuguaglianze (37.9) e (37.10), segue che

$$\begin{aligned} |C| &\le \textstyle\sum_{S \in C^*} H(|S|) \\ &\le |C^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) , \end{aligned}$$

che dimostra il teorema. Rimane quindi solo da dimostrare la disuguaglianza (37.10). Per qualsiasi insieme $S \in \mathcal{F}$ e $i = 1, 2, \ldots, |C|$, sia

$u_i = |S - (S_1 \cup S_2 \cup \ldots \cup S_i)|$

il numero degli elementi in S ancora non coperti dopo che $S_1, S_2, \ldots, S_i$ sono stati selezionati dall'algoritmo. Si definisce $u_0 = |S|$ il numero degli elementi di S, tutti inizialmente non coperti. Sia k il più piccolo indice tale che $u_k = 0$, così che ogni elemento in S è coperto da almeno uno degli insiemi $S_1, S_2, \ldots, S_k$. Allora, $u_{i-1} \ge u_i$ e $u_{i-1} - u_i$ elementi di S sono coperti per la prima volta da S_i, per $i = 1, 2, \ldots, k$. Quindi

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

Si osservi che

$$\begin{aligned} |S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| &\ge |S - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})| \\ &= u_{i-1}, \end{aligned}$$

dato che la scelta greedy di S_i garantisce che S non copra più elementi di quanti ne copra S_i (altrimenti, sarebbe stato scelto S al posto di S_i).

Conseguentemente si ottiene

$$\sum_{x \in S} c_x \le \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} .$$

Per valori interi a e b, con $a < b$, si ha

$$\begin{aligned} H(b) - H(a) &= \sum_{i=a+1}^{b} 1/i \\ &\ge (b-a)\frac{1}{b} . \end{aligned}$$

Usando questa disuguaglianza, si ottiene la somma telescopica

$$\begin{aligned} \sum_{x \in S} c_x &\le \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) \\ &= H(u_0) - H(0) \\ &= H(u_0) \\ &= H(|S|) , \end{aligned}$$

dato che $H(0) = 0$. Questo completa la dimostrazione della disuguaglianza (37.10). ■

Corollario 37.5.

GREEDY-SET-COVER ha rapporto limite $\ln|X| + 1$.

Dimostrazione. Si usi la disuguaglianza (3.12) ed il Teorema 37.4. ■

In alcune applicazioni $\max\{|S| : S \in \mathcal{F}\}$ è una costante piccola, e così la soluzione fornita da GREEDY-SET-COVER è più grande di una soluzione ottima al più per una costante moltiplicativa piccola. Ciò accade per esempio quando questa euristica è usata per ottenere una copertura di vertici approssimata, per un grafo i cui vertici hanno al più grado 3. In questo caso, la soluzione trovata con GREEDY-SET-COVER non è più di $H(3) = 11/6$ volte più grande di una soluzione ottima, una prestazione che è leggermente migliore di quella di APPROX-VERTEX-COVER.

Esercizi

37.3-1 Si considerino ciascuna delle seguenti parole come un insieme di lettere: {`arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread`}.

Si mostri quale insieme di copertura restituisce GREEDY-SET-COVER quando la parità è risolta in favore della parola che viene prima nel dizionario.

37.3-2 Si mostri che la versione decisionale del problema della copertura di un insieme è NP-completo per riduzione dal problema della copertura di vertici.

37.3-3 Si mostri come realizzare GREEDY-SET-COVER in modo tale che sia eseguito in tempo $O(\sum_{S \in \mathcal{F}} |S|)$.

37.3-4 Si mostri che la seguente forma più debole del Teorema 37.4 è banalmente vera: $|C| \le |C^*| \max\{|S| : S \in \mathcal{F}\}$.

37.3-5 Si crei una famiglia di istanze per la copertura di un insieme che dimostri che GREEDY-SET-COVER può restituire un numero di soluzioni diverse che è esponenziale rispetto alla dimensione dell'istanza. (Le soluzioni diverse discendono da differenti risoluzioni della parità nella scelta di S nella riga 4).

37.4 Il problema della somma di sottoinsieme

Un'istanza del problema della somma di sottoinsieme è un coppia (S, t), dove S un insieme $\{x_1, x_2, \ldots, x_n\}$ di interi positivi e t è un intero positivo. Questo problema decisionale chiede se esiste un sottoinsieme di S tale che la somma dei suoi elementi sia esattamente t. Questo problema è NP-completo (si veda il paragrafo 36.5.3).

Il problema di ottimizzazione associato a questo problema decisionale si manifesta in molte applicazioni pratiche. Nel problema di ottimizzazione si desidera trovare un sottoinsieme di $\{x_1, x_2, \ldots, x_n\}$ la cui somma sia la più grande possibile ma non più grande di t. Per esempio, si supponga di avere un vagone che non può trasportare più di t tonnellate ed n diversi pacchi da spedire, l'i-esimo dei quali ha peso x_i. Si desidera riempire il vagone il più possibile senza eccedere il peso limite dato.

In questo paragrafo, si presenta un algoritmo con tempo esponenziale per questo problema di ottimizzazione e quindi si mostra come modificare l'algoritmo per farlo diventare uno schema di approssimazione pienamente polinomiale. (Si ricordi che uno schema di approssimazione pienamente polinomiale ha un tempo di esecuzione che è polinomiale in $1/\varepsilon$ oltre che in n.)

Un algoritmo con tempo esponenziale

Se L è una lista di interi positivi ed x è un altro intero positivo, allora si indica con $L + x$ la lista di interi ottenuta aumentando di x ciascun elemento di L. Per esempio se $L = \langle 1, 2, 3, 5, 9 \rangle$, allora $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. Questa notazione sarà usata anche per gli insiemi, così che

$$S + x = \{s + x : s \in S\}.$$

Useremo una procedura ausiliaria MERGE-LISTS(L, L') che restituisce la lista ordinata ottenuta dalla fusione delle due liste ordinate di input L ed L'. Come la procedura MERGE usata per il merge sort (paragrafo 1.3.1), MERGE-LISTS è eseguita in tempo $O(|L| + |L'|)$. (Omettiamo di dare lo pseudocodice per MERGE-LISTS.) La procedura EXACT-SUBSET-SUM prende come input l'insieme $S = \{x_1, x_2, \ldots, x_n\}$ ed il valore obiettivo t.

```
EXACT-SUBSET-SUM(S, t)
1  n ← |S|
2  L0 ← ⟨0⟩
3  for i ← 1 to n
4      do Li ← MERGE-LISTS(Li-1, Li-1 + xi)
5          rimuovi da Li ogni elemento che è più grande di t
6  return l'elemento più grande in Ln
```

Si denoti con P_i l'insieme di tutti i valori che possono essere ottenuti selezionando un sottoinsieme (anche vuoto) di $\{x_1, x_2, \ldots, x_i\}$ e sommando i suoi elementi. Per esempio, se $S = \{1, 4, 5\}$, allora

$$P_1 = \{0, 1\},$$
$$P_2 = \{0, 1, 4, 5\},$$
$$P_3 = \{0, 1, 4, 5, 6, 9, 10\}.$$

Data l'identità

$$P_i = P_{i-1} \cup (P_{i-1} + x_i), \tag{37.11}$$

si può dimostrare per induzione su i (si veda l'Esercizio 37.4-1) che la lista L_i è una lista ordinata contenente tutti gli elementi di P_i il cui valore non è più grande di t. Poiché la lunghezza di L_i può raggiungere 2^i, EXACT-SUBSET-SUM è in generale un algoritmo con tempo esponenziale, anche se risulta con tempo polinomiale nel caso particolare in cui t sia polinomiale in $|S|$ oppure tutti i valori di S siano limitati da un polinomio in $|S|$.

Uno schema di approssimazione pienamente polinomiale

Si può ottenere uno schema di approssimazione pienamente polinomiale per il problema della somma di sottoinsieme "setacciando" ciascuna lista L_i dopo la sua creazione. Si usa un parametro di setaccio δ tale che $0 < \delta < 1$. Il *setaccio* di una lista L con un valore δ consiste nel rimuovere il maggior numero possibile di elementi da L in modo tale che se L' è il risultato del setaccio di L, allora per ogni elemento y che è stato rimosso da L, c'è un elemento $z \le y$ ancora in L' tale che

$$\frac{y - z}{y} \le \delta$$

o, equivalentemente,

$$(1 - \delta)y \le z \le y.$$

Si può pensare a un tale z come a un "rappresentante" di y nella nuova lista L'. Ciascun y è rappresentato da uno z tale che l'errore relativo di z, rispetto a y, è al più δ. Per esempio, se $\delta = 0.1$ e

$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle$,

allora si può setacciare L ottenendo

$L' = \langle 10, 12, 15, 20, 23, 29 \rangle$,

dove il valore cancellato 11 è rappresentato dal 10, i valori cancellati 21 e 22 sono rappresentati dal 20 ed il valore cancellato 24 è rappresentato dal 23. È importante ricordare che ogni elemento della lista setacciata è anche un elemento della versione originale della lista. Il setaccio di una lista può ridurre sensibilmente il numero degli elementi nella lista prendendo un valore rappresentante vicino (e leggermente più piccolo) per ciascun elemento cancellato dalla lista.

La seguente procedura setaccia una lista di input $L = \langle y_1, y_2, \ldots, y_m \rangle$ in tempo $\Theta(m)$, assumendo che L sia ordinata in ordine non decrescente. L'output della procedura è una lista ordinata setacciata.

```
TRIM(L, δ)
m ← |L|
L' ← ⟨y1⟩
last ← y1
for i ← 2 to m
    do if last < (1 − δ)yi
          then aggiungi yi alla fine di L'
               last ← yi
return L'
```

Gli elementi di L sono scanditi in ordine crescente, ed un numero è inserito nella lista L' restituita solo se è il primo elemento di L oppure non può essere rappresentato dal numero più recentemente inserito in L'.

Data la procedura TRIM, si può definire lo schema di approssimazione nel modo seguente. Questa procedura prende in input un insieme $S = \{x_1, x_2, \ldots, x_n\}$ di n interi (in un qualsiasi ordine), un valore obiettivo t, ed un "parametro di approssimazione" ε, con $0 < \varepsilon < 1$.

```
APPROX-SUBSET-SUM(S, t, ε)
1 n ← |S|
2 L0 ← ⟨0⟩
3 for i ← 1 to n
4     do Li ← MERGE-LISTS(Li−1, Li−1 + xi)
5        Li ← TRIM(Li, ε/n)
6        rimuovi da Li ogni elemento che è più grande di t
7 sia z il valore più grande in Ln
8 return z
```


La linea 2 inizializza la lista L_0 con il solo elemento 0. Il ciclo nelle linee 3-6 ha l'effetto di calcolare L_i, che è una lista ordinata contenente una versione opportunamente setacciata dell'insieme P_i, con tutti gli elementi più grandi di t rimossi. Poiché L_i è creata da L_{i-1}, si deve essere sicuri che setacci ripetuti non introducano un'eccessiva approssimazione. Tra poco si vedrà che APPROX-SUBSET-SUM restituisce una approssimazione corretta, se questa esiste.

Come esempio si supponga di avere l'istanza

$L = \langle 104, 102, 201, 101 \rangle$

con $t = 308$ ed $\varepsilon = 0.20$. Il parametro di setaccio δ è $\varepsilon/4 = 0.05$. APPROX-SUBSET-SUM calcola i seguenti valori nelle linee indicate:

linea 2:	L_0	=	$\langle 0 \rangle$,
linea 4:	L_1	=	$\langle 0, 104 \rangle$,
linea 5:	L_1	=	$\langle 0, 104 \rangle$,
linea 6:	L_1	=	$\langle 0, 104 \rangle$,
linea 4:	L_2	=	$\langle 0, 102, 104, 206 \rangle$,
linea 5:	L_2	=	$\langle 0, 102, 206 \rangle$,
linea 6:	L_2	=	$\langle 0, 102, 206 \rangle$,
linea 4:	L_3	=	$\langle 0, 102, 201, 206, 303, 407 \rangle$,
linea 5:	L_3	=	$\langle 0, 102, 201, 303, 407 \rangle$,
linea 6:	L_3	=	$\langle 0, 102, 201, 303 \rangle$,
linea 4:	L_4	=	$\langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
linea 5:	L_4	=	$\langle 0, 101, 201, 302, 404 \rangle$,
linea 6:	L_4	=	$\langle 0, 101, 201, 302 \rangle$.

L'algoritmo restituisce $z = 302$ che è ampiamente compreso nello scarto $\varepsilon = 20\%$ dalla soluzione ottima $307 = 104 + 102 + 101$; in effetti lo scarto è inferiore al 2%.

Teorema 37.6

APPROX-SUBSET-SUM è uno schema di approssimazione pienamente polinomiale per il problema della somma di sottoinsieme.

Dimostrazione. Le operazioni di setaccio di L_i nella linea 5 e di rimozione da L_i di ogni elemento più grande di t mantengono la proprietà che ogni elemento di L_i è anche elemento di P_i. Perciò, il valore z restituito nella linea 8 è indubbiamente la somma di un qualche sottoinsieme di S. Rimane da dimostrare che non sia più piccolo di $1 - \varepsilon$ moltiplicato per una soluzione ottima. (Si noti che il problema della somma di sottoinsieme è un problema di massimizzazione, e quindi l'equazione (37.2) è equivalente a $C'(1 - \varepsilon) \le C$.) Si deve dimostrare anche che l'algoritmo è eseguito in tempo polinomiale.

Per mostrare che l'errore relativo della risposta restituita è piccolo, si noti che quando la lista L_i è setacciata, si introduce un errore relativo di al più ε/n tra i valori rappresentativi che

rimangono ed i valori prima del setaccio. Per induzione su i, si può mostrare che per ogni elemento y in P_i che sia al più t, c'è uno $z \in L_i$ tale che

$$(1 - \varepsilon/n)^i y \le z \le y . \tag{37.12}$$

Se $y^* \in P_n$ denota una soluzione ottima del problema della somma di sottoinsieme, allora c'è uno $z \in L_n$ tale che

$$(1 - \varepsilon/n)^n y^* \le z \le y^* ; \tag{37.13}$$

il più grande di questi z è il valore restituito da APPROX-SUBSET-SUM. Dato che si può mostrare che

$$\frac{d}{dn}\left(1-\frac{\varepsilon}{n}\right)^n > 0 ,$$

la funzione $(1 - \varepsilon/n)^n$ cresce con n, per cui $n > 1$ implica

$$(1 - \varepsilon) < (1 - \varepsilon/n)^n,$$

e quindi

$$(1 - \varepsilon)y^* \le z .$$

Perciò il valore z restituito da APPROX-SUBSET-SUM non più piccolo di $1 - \varepsilon$ volte la soluzione ottima y^*.

Per dimostrare che questo è uno schema di approssimazione pienamente polinomiale si ricava una limitazione sulla lunghezza di L_i. Dopo il setaccio, elementi successivi z e z' di L_i devono soddisfare la relazione $z/z' > 1/(1 - \varepsilon/n)$. Cioè, essi devono differire per un fattore di almeno $1/(1 - \varepsilon/n)$. Perciò il numero di elementi in ogni L_i è al più

$$\log_{1/(1-\varepsilon/n)} t = \frac{\ln t}{-\ln(1-\varepsilon/n)} \le \frac{n \ln t}{\varepsilon} ,$$

usando l'equazione (2.10). Questo limite è polinomiale rispetto al numero n di valori dati in input, al numero di bit $\lg t$ necessari a rappresentare t e a $1/\varepsilon$. Poiché il tempo di esecuzione di APPROX-SUBSET-SUM è polinomiale rispetto alla lunghezza di L_i, APPROX-SUBSET-SUM è uno schema di approssimazione pienamente polinomiale. ■

Esercizi

37.4-1 Si dimostri l'equazione (37.11).

37.4-2 Si dimostrino le equazioni (37.12) e (37.13).

37.4-3 Come si potrebbe modificare lo schema presentato in questo paragrafo per trovare una buona approssimazione del più piccolo valore non minore di t che sia somma di un qualche sottoinsieme della lista data in input?

Problemi

37-1 Riempimento di contenitori

Si supponga di avere un insieme di n oggetti, tali che la dimensione s_i dell'oggetto i-esimo soddisfi $0 < s_i < 1$. Si vogliono mettere tutti questi oggetti nel minimo numero di contenitori di capacità unitaria. Ciascun contenitore può contenere qualsiasi sottoinsieme degli oggetti la cui dimensione totale non superi 1.

a. Si dimostri che il problema della determinazione del minimo numero di contenitori richiesti è NP-arduo. (*Suggerimento*: ci si riconduca al problema della somma di sottoinsieme.)

L'euristica *first-fit* prende ciascun oggetto uno dopo l'altro e lo pone nel primo contenitore che lo può contenere. Sia $S = \sum_{i=1}^{n} s_i$.

b. Si mostri che il numero ottimo di contenitori richiesti è almeno $\lceil S \rceil$.

c. Si mostri che l'euristica first-fit lascia al più un contenitore pieno per meno di metà.

d. Si dimostri che il numero di contenitori usati dalla euristica first-fit non è mai più di $\lceil 2S \rceil$

e. Si dimostri che l'euristica first-fit ha rapporto limite 2.

f. Si fornisca una realizzazione efficiente della strategia first-fit e si analizzi il suo tempo di esecuzione.

37-2 Approssimazione della dimensione di una cricca massima

Sia $G = (V, E)$ un grafo non orientato. Per qualsiasi $k \ge 1$, si definisce $G^{(k)}$ il grafo non orientato $(V^{(k)}, E^{(k)})$, dove $V^{(k)}$ è l'insieme di tutte le k-uple ordinate di vertici di V, e $E^{(k)}$ è definito in modo che $(v_1, v_2, \ldots, v_k)$ sia adiacente a $(w_1, w_2, \ldots, w_k)$ se e solo se, per ogni i con $1 \le i \le k$, o il vertice v_i è adiacente a w_i in G, oppure $v_i = w_i$.

a. Si dimostri che la dimensione della cricca più grande in $G^{(k)}$ è uguale alla k-esima potenza della dimensione della cricca più grande di G.

b. Si deduca che se esiste un algoritmo approssimato che ha rapporto limite costante per trovare una cricca di dimensione massima, allora per questo problema c'è uno schema di approssimazione pienamente polinomiale.

37-3 Il problema della copertura pesata di un insieme

Si supponga di generalizzare il problema della copertura di un insieme in modo che ciascun insieme S_i nella famiglia $\mathcal{F}$ abbia un peso w_i associato ed il peso di una copertura $\mathcal{C}$ sia $\sum_{S_i \in \mathcal{C}} w_i$. Si desidera determinare una copertura di peso minimo. (Il paragrafo 37.3 gestisce il caso in cui $w_i = 1$ per tutti gli i.)

Si mostri come l'euristica greedy per la copertura di un insieme possa essere facilmente generalizzata per offrire una soluzione approssimata per qualsiasi istanza del problema della copertura pesata di un insieme. Si mostri che una tale euristica ha rapporto limite $H(d)$, dove d è la dimensione massima di un qualsiasi insieme S_i.

Note al capitolo

Esistono molte pubblicazioni sugli algoritmi approssimati. Un testo introduttivo è quello di Garey e Johnson [79]. Anche Papadimitriou e Steiglitz [154] offrono un'eccellente presentazione degli algoritmi approssimati. Lawler, Lenstra, Rinnooy Kan e Shmoys [133] offrono una trattazione esauriente del problema del commesso viaggiatore.

Papadimitriou e Steiglitz attribuiscono l'algoritmo APPROX-VERTEX-COVER a F. Gavril e M. Yannakakis. L'algoritmo APPROX-TSP-TOUR appare in un'eccellente pubblicazione di Rosenkrantz, Stearns e Lewis [170]. Il Teorema 37.3 è dovuto a Sahni e Gonzalez [172]. L'analisi dell'euristica greedy per il problema della copertura di un insieme si basa sulla dimostrazione di Chvátal [42] di un risultato più generale; il risultato presentato nel capitolo è dovuto a Johnson [113] e Lovász [141]. L'algoritmo APPROX-SUBSET-SUM e la sua analisi sono variazioni degli algoritmi approssimati relativi ai problemi dello zaino e della somma di sottoinsieme, dovuti a Ibarra e Kim [111].

Bibliografia

[1] Milton Abramowitz e Irene A. Stegun, editor. *Handbook of Mathematical Functions*. Dover, 1965.

[2] G. M. Adel'son-Vel'skiĭl e E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259-1263, 1962.

[3] Leonard M. Adleman, Carl Pomerance e Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Mathematics*, 117:173-206, 1983.

[4] Alfred V. Aho, John E. Hopcroft e Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[5] Alfred V. Aho, John E. Hopcroft e Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[6] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin e Robert E. Tarjan. Faster algorithms for the shortest path problem. Technical Report 193, MIT Operations Research Center, 1988.

[7] Howard H. Aiken e Grace M. Hopper. The automatic sequence controlled calculator. In Brian Randell, editor, *The Origins of Digital Computers*, pp 203-222. Springer-Verlag, third edition, 1982.

[8] M. Ajtai, J. Komlós e E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pp 1-9, 1983.

[9] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989. Tradotto in lingua italiana da Jackson Libri - "Progettazione ed analisi degli algoritmi paralleli"

[10] Richard J. Anderson e Gary L. Miller. Deterministic parallel list ranking. In John H. Reif, editor, *1988 Aegean Workshop on Computing*, volume 319 of *Lecture Notes in Computer Science*, pp 81-90. Springer-Verlag, 1988.

[11] Richard J. Anderson e Gary L. Miller. A simple randomized parallel algorithm for list-ranking. Manoscritto non pubblicato, 1988.

[12] Tom M. Apostol. *Calculus*, volume 1. Blaisdell Publishing Company, seconda edizione, 1967. (*Calcolo*, volume 1, 1977, Boringhieri).

[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. Comunicazione privata. 1989.

[16] Eric Bach. Number-theoretic algorithms. In *Annual Review of Computer Science*, volume 4, pp 119-172. Annual Reviews, Inc., 1990.

[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. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pp 75-84. IEEE Computer Society, 1975.

[195] Uzi Vishkin. Implementation of simultaneous memory address access in models that forbit 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 e J. L. Smith. A one-microsecond adder using one-megacycle circuitry. *IRE Transactions on Electronic Computers*, EC-5(2), 1956.

[200] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 47:509-533, 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. In *Actes du Congrès International des Mathematiciens*, volume 3, pp 383-288, 1970.

[204] James C. Wyllie. *The Complexity of Parallel Computations*. 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.

Indice analitico

Simboli

3-FNC 886
3-soddisfattibilità 886
3-soddisfattibilità in forma normale congiuntiva 886
6-colorazione 683

A

abbinamento
  massimo bipartito 571
  perfetto 573
accessi concorrenti 652
Ackermann 432
addizionatore 624
  completo 621
  con propagazione del riporto 625
  senza riporto 631
  seriale 642
addizione
  con previsione del riporto 626, 631
  con propagazione del riporto 625, 643
  di polinomi 735
  modulare 771
  seriale 641, 643
  tra matrici 692
aggiornamento del flusso massimo 594
agglomerazione primaria 221
agglomerazione secondaria 221
alberi 84, 85, 202, 499
  2-3-4 367, 380
  albero di copertura minimo per grafi sparsi 488
  altezza di un nodo 134
  ammortizzati con peso bilanciato 358
  antenato 87
  B-alberi 366
  BFS 456
  binari 88, 201
  binari di ricerca 229, 232, 238, 243
  binomiali 384
  binomiali non ordinati 406
  bisezione 91
  costruzione di un albero di copertura minimo 478
  DFS 458
  di cammini minimi 494, 499
  di copertura 328
  di copertura (il più piccolo dopo il minimo) 488
  di decisione 164
  di intervalli 272
  di parsing 303
  di selezione 263, 265
  diametro 457
  discendente 87
  foresta DFS 458
  lessicografici 243
  liberi 85
  mantenimento della dimensione dei sottoalberi 266
  nodo 87
  ordinati 87
  padre 87
  posizionali 88
  proprietà degli alberi liberi 85
  radicati 87, 201
  radice 87
  RB-alberi 247
  ricorsione 62
  sottoalbero radicato in x 87
albero binario
  b-altezza 247
  bilanciato 247
  colore 247
  di ricerca 229
  interrogazioni su un albero binario di ricerca 231
  proprietà RB 247
  visita in ordine anticipato 230
  visita in ordine differito 230
  visita in ordine simmetrico 230

albero di copertura minimo 478, 488, 549
visita completa 913
alfabeto 867
alfabeto di input 813
algoritmi 1, 10, 12
analisi 5, 12
approssimati 907
Bellman-Ford 507, 519
Boyer-Moore 825
con tempo esponenziale 920
confronto tra algoritmi EREW e algoritmi CRCW 663
contadino russo 643
correttezza del metodo dei preflussi 578
di Tarjan per i minimi antenati comuni off-line 442
Dijkstra 503
divide-et-impera 855
Edmonds-Karp 567
ellissoide 514
etichette di cammini orientati mediante programmazione dinamica 547
Euclide 766, 767, 768, 802
Floyd-Warshall 532
Ford-Fulkerson 564
Graham 847
greedy 311
greedy approssimato 916
heapsort 139
incrementale di Gabow per cammini minimi 520
ingenuo di corrispondenza tra stringhe 806
intersezione tra segmenti 842
Jarvis 852
Johnson per grafi sparsi 539
Karmarlar 514
Karp per il ciclo di peso medio minimo 521
Knuth-Morris-Pratt 819
Kruskal e Prim 482
lift-to-front 583, 588
massimo efficiente rispetto al numero di processori 685
MCD binario 801
minimo albero di copertura con gli heap aggregabili 400
paralleli 651
partizionamento di Lomuto 158
preflusso 574, 578, 580
progetto 10
Rabin-Karp 809
riduzione 875
simplesso 513
simulazione di un algoritmo CRCW con un algoritmo EREW 667
Stooge Sort 159
Strassen per la moltiplicazione tra matrici 697
verifica 872
Viterbi 308
algoritmi greedy
analisi comparata degli algoritmi greedy e della programmazione dinamica 316
dimostrazione della correttezza 314
dimostrazione della correttezza dell'algoritmo greedy 314
fondamenti teorici dei metodi greedy 326
proprietà della scelta greedy 316
selezione di attività 312
sottostruttura ottima 316
strategia 315
su un matroide pesato 328
algoritmi paralleli 651
PRAM 651
algoritmo approssimato
errore relativo 908
schema di approssimazione 908
allocazione e deallocazione di oggetti 198
altezza 134
ammissibilità 514
analisi
comparata degli algoritmi greedy 316
delle operazioni sui bit dell'algoritmo di Euclide 802
heap d-ari 144
organizzazione hash con 211
procedura Ford-Fulkerson 566
programma max 124
unione per rango con compressione 432
analisi ammortizzata 337
metodo degli accantonamenti 342
metodo degli aggregati 338
metodo del potenziale 344
analisi probabilistica
paradosso del compleanno 118
AND 620
anelli 705
annidamento degli intervalli dei discendenti 462
annullatore 543, 704
antenato proprio 87
antisimmetria 552
approccio incrementale 10
approssimazione della dimensione di una cricca massima 925
approssimazioni 47
arbitraggio 520
arco 81, 478, 492

adiacente 81
dell'albero 456
etichetta 544
incidente 81
leggero 479
pesi negativi 492
residuo 559
reti ammissibili 584
argomento 79
aritmetica modulare 770
associatività 770
chiusura 770
identità 770
opposto 770
reciproco 770
array 16, 146
analisi del programma max 124
intero mancante 69
inversioni 16
multipli 197
partizionamento 146
rappresentazione di oggetti con array multipli 197
valore massimo 124
assegnamento di soddisfattibilità 878, 884
assegnamento di verità 878, 884
assiomi della probabilità 98
associatività 74
assorbimento 74
attributi 4
automi 814
automi a stati finiti 813
alfabeto di input 813
funzione di transizione 813
funzione stato-finale 813
automi per la corrispondenza tra stringhe 814

B

B-alberi 366, 367
altezza 367
creazione di un B-albero vuoto 370
definizione 366
divisione di un nodo in un B-albero 370
eliminazione di una chiave 376
inserimento di una chiave 372
operazioni di base 369
ricerca 369
b-altezza 247
base 29, 66
Batcher, reti di fusione pari-dispari 616
Bayes, teorema 101
Bellman-Ford 507, 509
Bernoulli 107
bilanciato 247
bisezione di alberi 91
bit di riporto di ingresso 625
bit di somma 625
biunivoca 80
Boole 102
Boyer-Moore 825
Brent 670
Bucket sort 170

C

calcolabile in tempo polinomiale 866
calcolo
bottom-up dei pesi di cammino minimo 528, 534
degli stati del riporto con un circuito parallelo per il calcolo dei prefissi 628
dei cammini minimi tra tutte le coppie 541
dei costi ottimi 287
dei prefissi 627, 658, 685
dei ranghi 655
della funzione di transizione 818
della lunghezza di una LCS 298
dell'inversa di una matrice da una fattorizzazione LUP 720
di un MIS da una 6-colorazione 684
di una 6-colorazione 683
incrementale del flusso massimo 594
calcolo combinatorio 93
coefficienti binomiali 95
combinazioni 94
limiti binomiali 95
permutazioni 94
regole di somma e prodotto 93
stringhe 94
calcolo delle probabilità 98
assiomi 98
contatore probabilistico 125
distribuzione binomiale 107
distribuzione continua uniforme 99
distribuzione discreta di probabilità 99
distribuzione geometrica 107
probabilità condizionata ed indipendenza 100
valore atteso di una variabile casuale 104
variabili casuali discrete 103
varianza e scarto quadratico medio 106
calcolo parallelo 658, 674
cammini 593
concatenazione 544
cammini aumentanti 561
cammini minimi 453, 491, 493, 495, 499, 527, 528, 534, 541
alberi 499

algoritmo incrementale di Gabow per cammini minimi 520
calcolo bottom-up dei pesi di cammino minimo 528, 534
calcolo dei cammini minimi tra tutte le coppie 541
moltiplicazione di matrici 527
peso di cammino minimo 491
sorgente singola in grafi orientati aciclici 511
sottostruttura ottima 495
struttura 527, 533
vincoli di differenza e cammini minimi 513
cammino 82
etichetta 544
cammino critico 512
cammino di accesso 429
cammino hamiltoniano 874
campi 4, 707
cancellazione 255
albero binario di ricerca 235
lista concatenata 194
capacità residua 559, 561
cappi 81
cardinalità 75
Carmichael 793
catena destra 853
catena sinistra 853
certificati 790
chiave 141
mediana 370
pubblica 785
segreta 785
chiusura 868
chiusura transitiva di un grafo dinamico 549
chiusura transitiva di un grafo orientato 536
ciclico 782
ciclo 82
di clock 641
di Eulero 475
di peso medio minimo 521
euleriano 660
hamiltoniano 871, 897
circuiti
di divisione 647
moltiplicatori 634
tally 634
temporizzati 641
circuiti combinatori 619, 621
addizionatore completo 621
addizionatore senza riporto 631
addizionatori 624
addizione con previsione del riporto 626, 631
addizione con propagazione del riporto 625
circuiti moltiplicatori 634
dimensione 623
moltiplicatori a griglia 635
moltiplicatori ad albero di Wallace 638
profondità 623
circuiti paralleli, calcolo dei prefissi 628
circuiti sequenziali 641
addizione seriale 641
moltiplicatori a catena 643
classe di complessità 869
co-NP 873
NP 872
P 866
classe di congruenza modulo n 761
classe di equivalenza 77
classificazione degli archi 462
classificazione degli archi con visita in ampiezza 474
clock 641
coda 189, 190
coda 190
con priorità 141
della distribuzione binomiale 113
testa 190
codici
a lunghezza fissa 319
a lunghezza variabile 319
di Huffman 319
prefissi 320
codifica 865
correlate polinomialmente 866
codominio 79
coefficienti 735
coefficienti binomiali 95
cofattore 695
collisione 209
collisioni, risoluzione per concatenazione 210
colorazione 680
colorazione di un grafo 90, 904
colorazioni e insiemi indipendenti massimali 680
colore 247
colpo mancato 810
combinazione convessa 835
combinazioni 94
commesso viaggiatore 911
commesso viaggiatore bitonico e euclideo 306
commutatività 74
commutativo 543
complemento 74, 99, 868, 892
complemento di Schur 714, 725
componenti connesse 686
componenti fortemente connesse 468
compressione dei cammini 430

concatenazione 544, 806, 868
configurazione 880
confrontabili 841
confronto tra liste 203
confronto tra funzioni 27
riflessività 27
simmetria 27
simmetria trasposta 27
transitività 27
tricotomia 28
congruente 761
congruenza modulare 761
connettività per archi 570
conservazione del flusso 552
contadino russo algoritmo 643
contatore binario 340
dei bit rovesciati 356
contatore del programma 880
contatore probabilistico 125
controllo 653
convoluzione 737, 748
copertura
di vertici 892, 909
minima 593
minima con cammini 593
ottima di vertici 909
pesata di un insieme 925
coppia di segmenti 840
coppia ordinata 76
coppie punto/valore 737
correttezza
del calcolo della funzione prefisso 822
del metodo dei preflussi 578
della visita in ampiezza 455
correttezza dell'algoritmo
di Dijkstra 503
di Graham 850
di Huffman 323
generico di preflusso 579
greedy sui matroidi 331
KMP 824
correttezza di RSA 789
corrispondenza tra stringhe 805, 814
basata sui fattori di ripetizione 832
con gli automi a stati finiti 813
su un calcolatore parallelo 832
corrispondenza uno-a uno 80
costante di Eulero-Mascheroni 43
costo ammortizzato 338, 342
costo dei calcoli aritmetici 759
costruzione
di un albero di copertura minimo 478
di un cammino minimo 534
di un codice di Huffman 321
di una LCS 299
di uno heap 137
di uno heap con l'inserzione 143
counting sort 166
CRCW 653, 663, 667, 686
creazione di un B-albero vuoto 370
creazione di un nuovo heap binomiale 389
credito 342
crescita asintotica 34
CREW 653
crittografia 785
crittografia a chiave pubblica RSA 785

D

dag 84
decremento di una chiave 397
DeMorgan, leggi 74
densità dei numeri primi 792
deque 192
derivate 41
determinante 694, 695
determinazione della profondità 441
determinazione della svolta a sinistra o a destra 837
DFT 742, 744, 745
diagramma di Venn 74
diagrammi PERT 512
diametro 457
differenza 74
Dijkstra 503
dimensione 75
dei circuiti 623
dell'input 6, 759
dimostrazione
della correttezza dell'algoritmo greedy 314
di NP-completezza 883
discendente 87
discendente proprio 87
diseguaglianza della funzione suffisso 816
disgiunti 75
distanza
di editing 307
Manhattan 183
sul cammino minimo 453
distributività 74
distribuzione
binomiale 107
binomiale code 113
continua uniforme di probabilità 99
di probabilità 98
discreta di probabilità 99
geometrica 107
inviluppata sparsa 860

distribuzioni
  inviluppate sparse 860
disuguaglianza
  di Boole 102
  di Markov 107
  triangolare 911
divide-et-impera 10, 12, 855
divisibilità 760
divisione di un nodo in un B-albero 370
divisori 760
  banali 760
  comuni 762
dominio 79

E

Edmonds-Karp 567
effetto delle euristiche sul tempo di esecuzione 431
elementi 73
elementi combinatori 620, 670
elementi di programmazione dinamica 291
elemento neutro 704
elevamento a potenza 784
elevamento a potenza con quadrature ripetute 784
eliminazione
  di Gauss 713
  di una chiave 397
  di una chiave da un B-albero 376
ellissoide 514
equazione normale 728
equazioni
  lineari modulari 776
  notazione asintotica 25
  risoluzione di equazioni lineari modulari 776
ERCW 653
EREW 652, 663, 667
errore relativo 908
errori di approssimazione 727
espansione della tabella 349
espansione e contrazione di tabelle 352
esperimenti con navette spaziali 594
esponenziali 29
estendere una struttura di dati 268
estensione dalle potenze di un intero a tutti i reali 70
estensione di RB-alberi 269
estrazione del nodo con la chiave minima 395
estremi 836
etichette di cammini orientati mediante programmazione dinamica 547
Euclide 766, 768
Eulero 43, 475, 773, 782
euristica 431
  del buon suffisso 828
  del carattere discordante 827
  del punto più vicino 915
  euristica rho di Pollard 798
  per migliorare il tempo di esecuzione 429
eventi
  (mutuamente) indipendenti 101
  certi 98
  elementari 98
  indipendenti 100
  indipendenti a coppie 101
  indipendenti in modo condizionato 103
  mutuamente esclusivi 98
  nullo 98

F

fantasmi ed acchiappafantasmi 859
fattore di carico 211, 348
fattori primi 797
fattoriali 31
fattorizzazione LU 713
fattorizzazione LUP 709, 715
Fermat 782
FFT 742, 745, 749, 753, 756
Fibonacci 420
  creazione di un nuovo heap di Fibonacci 406
  heap 404
  numeri 70, 802
FIFO 189
firma numerica 787
Floyd-Warshall 532
flusso 552
  massimo con capacità superiori ed inferiori 595
  netto 553, 562
  netto positivo 553
FNC 886
FND 889
fondamenti teorici dei metodi greedy 326
Ford-Fulkerson 558, 564, 566
foreste 84
  DFS 458
  di insiemi disgiunti 429
forma
  canonica 332
  early-first 332
  estesa dell'algoritmo di Euclide 768
  normale 3-congiuntiva 886
  normale congiuntiva 886
  normale disgiuntiva 888
formula di Lagrange 738
formule Booleane 648
  funzioni simmetriche 648
Fourier 745

funzioni 79
  argomento 79
  biunivoche 80
  codominio 79
  comuni 29
  confronto 27
  corrispondenza uno-a uno 80
  dominio 79
  formule Booleane per funzioni simmetriche 648
  funzione altezza 575
  funzione booleana 96
  funzione del buon suffisso 830
  funzione dell'ultima occorrenza 828
  funzione di Ackermann e la sua inversa 432
  funzione di densità 104
  funzione di densità congiunta 104
  funzione di distribuzione di probabilità 173
  funzione di entropia 96
  funzione di etichettatura 543
  funzione di maggioranza 648
  funzione di riduzione 875
  funzione fi di Eulero 773
  funzione generatrice 70
  funzione hash 209, 790
  funzione lineare 8
  funzione obiettivo 513
  funzione peso 448
  funzione potenziale 344
  funzione quadratica 8
  immagine 80
  iniettive 80
  inversa 80
  iterate 36
  permutazione 80
  rango 80
  sequenza finita 79
  sequenza infinita 79
  simmetriche 648
  surgettive 80
  valore 79
funzioni hash 213
  hashing doppio 221
  indirizzamento aperto 219
  metodo di divisione 215
  metodo di moltiplicazione 215
  scansione lineare 221
  scansione quadratica 221
  universale 216
fusione 611

G

Gabow 520
garbage collector 198
Gauss 713
generatore 782
generatore di numeri casuali 152
generatore di numeri pseudo-casuali 152
generazione di pesi non negativi tramite la rideterminazione dei pesi 540
Giuseppe Flavio 276
grado
  entrante 82
  limite 735
  minimo 367
  uscente 82
grafi 81, 515, 536
  abbinamento massimo in un grafo bipartito 570
  aciclici 82
  albero 84
  algoritmo di Johnson per grafi sparsi 539
  "amichevoli" 91
  annidamento degli intervalli dei discendenti 462
  archi 81
  arco adiacente 81
  arco incidente 81
  cammini minimi 453
  cammino 82
  cammino semplice 82
  cappi 81
  chiusura transitiva di un grafo dinamico 549
  chiusura transitiva di un grafo orientato 536
  ciclo 82
  classificazione degli archi 462
  colorazione 90
  completi 84
  componenti connesse 82, 687
  componenti fortemente connesse 83, 468
  connessi 82
  connessi singolarmente 465
  d-regolari 574
  dag 84
  delle componenti 473
  densi 447
  di vincoli 515
  dinamici 549
  diretti 81
  distanza sul cammino minimo 453
  foresta 84
  fortemente connessi 83
  generazione di pesi non negativi tramite la rideterminazione dei pesi 540
  grado entrante 82
  grado uscente 82
  grafo bipartito 84
  grafo completo 84
  grafo non orientato 81
  indotti 83

insieme degli archi 81
insieme dei vertici 81
iperarco 84
ipergrafo 84
isomorfi 83
matrice di incidenza 450
multigrafo 84
non orientati 81
ordinamento topologico 465
orientati 81, 543
pesati 448
punti di articolazione, ponti e componenti biconnesse 474
rappresentazione 447
rappresentazione con liste di adiacenza 447
rappresentazione con matrice di adiacenza 448
ricerca di un abbinamento massimo bipartito 571
sottocammino 82
sottografo 83
sparsi 447, 539
versione non orientata 83
versione orientata 83
vertice isolato 82
vertici 81
visita in ampiezza 450, 453, 455, 474
visita in profondità 458, 460
grafo bipartito 84
abbinamento massimo 570
Graham 847
gruppo
abeliano 770
additivo modulo n 771
finito 770
moltiplicativo modulo n 772

H

half-cleaner 608
Hall 574
hash 209
agglomerazione primaria 221
agglomerazione secondaria 221
collisione 209
fattore di carico 211, 348
funzione hash one-way 790
funzioni 213
hashing doppio 221
hashing universale 216
metodo di concatenazione 210
sequenza di scansione 219
tabelle 209
uniformità della funzione hash 220
uniformità semplice della funzione hash 211
hashing
doppio 221
k-universale 228
universale 216
heap 133, 400
2-3-4 399
aggregabili 383
costruzione 137
heap di Fibonacci 404
mantenimento della proprietà 135
minimo albero di copertura con gli heap aggregabili 400
operazioni che fondono heap 406
proprietà di ordinamento parziale 134
heap binomiali 384, 386
creazione di un nuovo heap 389
decremento di una chiave 397
eliminazione di una chiave 397
estrazione del nodo con la chiave minima 395
inserimento di un nodo 395
operazioni 389
rappresentazione 387
ricerca della chiave minima 389
unione di due heap binomiali 390
heap di Fibonacci 404, 420
creazione di un nuovo heap di Fibonacci 406
decremento di una chiave 413
eliminazione di un nodo 413, 416, 420
estrazione del nodo minimo 408
funzione potenziale 405
grado massimo 406
inserzione di un nodo 407
limitazione del grado massimo 417
operazioni 420
ricerca del nodo minimo 408
unione di due heap di Fibonacci 408
Horner 737
Huffman 319
codici 319
codici prefissi 320
correttezza dell'algoritmo di Huffman 323
costruzione di un codice di Huffman 321

I

idempotente 543
idempotenza 74
immagine 80
immagine raster 688
impacchettamento 852
impulso di clock 641
incapsulamento di scatole 519
incrementale 10

incremento di un contatore binario 340, 343, 346
indice 782
indirizzamento aperto 219
indirizzamento diretto 207
induzione matematica 43
informazione aggiunta 273
iniettiva 80
input
dimensione 6
inserimento
di un nodo 395
di una chiave in un B-albero 372
in una lista concatenata 194
insertion sort 2, 6
caso medio 8
caso peggiore 8
tempo di esecuzione 9
inserzione 251
albero binario di ricerca 235
e cancellazione 235
elementare 350
insiemi 73
associatività 74
assorbimento 74
cardinalità 75
colorazioni e insiemi indipendenti massimali 680
commutatività 74
complemento 74
definizione di semianelli chiusi 543
dimensione 75
disgiunti 75
distributività 74
finito 75
idempotenza 74
infinito 75
infinito numerabile 75
insieme degli archi 81
insieme dei vertici 81
insieme indipendente 680, 903
insieme k-separante 684
insieme parzialmente ordinato 78
insieme potenza 75
insieme vuoto 73, 74
insiemi dinamici persistenti 260
insiemi indipendenti massimali 680
interi 73
leggi di DeMorgan 74
massimali 78
minimo e massimo 175
minimo e massimo simultanei 176
non numerabile 75
numeri naturali 73
numeri reali 73
operazioni su insiemi disgiunti 423
partizione 75
prodotto cartesiano 76
proprietà dell'insieme vuoto 74
rappresentante 423
relazioni 77
selezione con tempo medio lineare 177
selezione in tempo lineare nel caso peggiore 179
selezione su un insieme dinamico 263
sequenza degli i numeri più grandi 182
sottoinsieme proprio 73
insiemi disgiunti 423
analisi dell'unione per rango con compressione dei cammini 432
euristica per l'unione pesata 426
euristiche per migliorare il tempo di esecuzione 429
foreste 429, 430
funzione di Ackermann 432
procedure per foreste di insiemi disgiunti 430
proprietà dei ranghi 435
rappresentazione a lista concatenata 426
unione 426
instabilità numerica 709
integrali 41, 47
interi 73
in fattori primi 797
primi tra loro 763
interpolazione 738, 747
interpolazione nelle radici complesse dell'unità 747
interrogazioni su un albero binario di ricerca 231
intersezione 74, 868
di una coppia qualsiasi 840
tra due segmenti 837
tra segmenti 842
intervallo chiuso 271
inversa 80, 694
inversione di matrici 720, 721
inversioni 16
inverso rispetto alla moltiplicazione 707
inverso rispetto all'addizione 705
inviluppo convesso 846
metodo divide-et-impera 846
metodo incrementale 846
metodo prune-and-search 846
invio non saturante 576
invio saturante 576
iperarco 84
ipergrafo 84
istanza di un problema 2
iterazione di Newton 647

J

Jarvis 852

K

k-combinazione 94
k-permutazione 94
k-sottoinsieme 75
k-sottostringa 94
k-stringa 94
Karmarlar 514
Karp 521
Kleene 868
Knuth-Morris-Pratt 819
KPG 631
Kruskal 482

L

Lagrange 738, 774
Lamé 768
Legendre 803
legge commutativa 770
leggi di DeMorgan 74
lemmi
  cancellazione 742
  complemento di Schur 725
  dimezzamento 743
  iterazione della funzione prefisso 823
  ricorsione della funzione suffisso 817
  sommatoria 744
  sovrapposizione dei suffissi 806
  strette di mano 84
lettura-concorrente 652
lettura-esclusiva 652
letture concorrenti 663
LIFO 189
lift-to-front 583
limitazione del grado massimo 417
limite
  alla dimensione della tabella
    nella concatenazione 227
  alla scansione più lunga in un'organizzazione
    hash 226
  di tempo 436
limiti
  binomiali 95
  inferiori sull'ordinamento per confronti nel caso
    medio 173
  stretti sulla sommatoria 157
linearità 40
linearmente dipendenti 694
linearmente indipendenti 694
linguaggi formali 867
linguaggio 867
  NP-arduo 876
  NP-completo 876
  verificato 872
  vuoto 868
lista
  bidirezionale 192
  circolare 193
  concatenata doppia 192
  dei figli 404
  dei vicini 585
  delle radici 388, 404
  libera 198
liste concatenate 192
  calcolo parallelo dei prefissi su una lista 658
  cancellazione 194
  confronti 203
  inserimento 194
  mergeable heap 204
  oggetti 197
  puntatori 197
  ricerca 193
  ricerca in una lista ordinata compatta 204
  sentinelle 194
logaritmi 30
logaritmo discreto 783
Lomuto, algoritmo di partizionamento 158
LU 713
LUP 709

M

macchina parallela ad accesso diretto 651
mantenimento della dimensione dei sottoalberi
  266
Markov 107
massimali 78, 327
massimo 175
  albero binario di ricerca 232
massimo comun divisore 762, 765
matching massimo in un grafo bipartito 570
matrici 285, 689, 695, 704
  addizione 692
  algoritmo di Strassen per la moltiplicazione
    697
  anelli 705
  calcolo dell'inversa di una matrice da una
    fattorizzazione LUP 720
  calcolo di una fattorizzazione LU 713
  campi 707
  compatibili 692
  definite positive 695
  dei predecessori 526

  determinante 695
  di incidenza 895
  di incidenza di un grafo non orientato 335
  di permutazione 691
  di Toeplitz 754
  di Vandermonde 697
  diagonale 690
  hermitiane 724
  identità 690
  inversa 694
  inverse, rango e determinante 694
  inversione di matrici 720
  invertibile 694
  minimi quadrati 724
  minore 695
  moltiplicazione tra matrici 692
  moltiplicazione tra matrici
    booleane 704, 706
  moltiplicazione tra matrici e inversione di
    matrici 721
  negativa 692
  non invertibile 694
  non-singolare 694
  norma (euclidea) 693
  nulla 690
  numero di parentesizzazioni 285
  operazioni sulle matrici 692
  parentesizzazione ottima 286
  presentazione della fattorizzazione LUP 709
  prodotto completamente parentesizzato 284
  prodotto di una sequenza 284
  prodotto esterno 693
  prodotto interno 693
  prodotto per uno scalare 692
  pseudoinversa 728
  quasianelli 704
  rango di colonna 694
  rango di riga 694
  simmetrica 692
  simmetriche 724
  singolare 694
  sistemi algebrici di numeri 704
  soluzione ricorsiva del problema del prodotto 286
  sottomatrice portante 724
  sottrazione tra matrici 692
  triangolare inferiore unitaria 691
  triangolare superiore unitaria 691
  tridiagonale 691
matroidi 326, 330, 331
  algoritmi greedy su un matroide pesato 328
  matroide grafico 326
  matroidi matriciali 326
  pesati 328
  proprietà di scambio 326
MCD binario 801
mediano 175
mediano pesato 182
membri 73
memoria
  accessi concorrenti 652
  primaria 364
  principale 364
  secondaria 364, 379
merge sort 13
mergeable heap 204
metodo
  degli accantonamenti 342
  dei minimi quadrati 726
  dei preflussi 574, 578, 580
  di concatenazione 210
  di divisione 215
  di Ford-Fulkerson 558
  di moltiplicazione 215
  di sostituzione 51, 52
  divide-et-impera 846
  incrementale 846
  iterativo 51, 55
  principale 51, 58
  prune-and-search 846
metodo degli accantonamenti
  incremento di un contatore binario 343
  operazioni su pile 342
metodo degli aggregati 338
  incremento di un contatore binario 340
  operazioni su pile 338
metodo del potenziale 344
  incremento di un contatore binario 346
  operazioni su pile 345
miglioramento di Yen all'algoritmo di Bellman-
  Ford 519
Miller-Rabin 793
minimi quadrati 724, 727
minimo 175
  albero binario di ricerca 232
  antenato comune 442
  comune multiplo 770
  e massimo 232
  e massimo simultanei 176
  off-line 440
MIS 6-colorazione 683
MIS da una 6-colorazione 683
misura di complessità 869
modelli PRAM 651
moltiplicatori 634
  a catena 643
  a griglia 635
  ad albero di Wallace 638

moltiplicazione
di matrici 527
di polinomi 736
divide-et-impera 754
modulare 771
tra matrici 692, 721
veloce di polinomi 740
moneta perfetta 99
monoide 543, 704
monotonicità 29
multigrafo 84

N

n-insieme 75
NAND 620
navette spaziali 594
negativa 692
Newton, iterazione per circuiti di divisione 647
nodo 87
nodo minimo 404
NOR 620
norma (euclidea) 693
NOT 620
notazione asintotica 21, 35
confronto 27
equazioni 25
funzioni iterate 36
notazione Ω 24
notazione O 23
notazione o 26
notazione Q 22
notazione ω 27
proprietà 35
notazione di sommatoria implicita 556
notazioni standard 29
base 29
e funzioni comuni 29
esponenziali 29
fattoriale 31
logaritmi 30
monotonicità 29
numeri di Fibonacci 33
polinomi 29
tetto 29
NP-completezza 875, 876, 883
NP-completi 863
nucleo 854
numeri
Catalani 285
composti 760
di Carmichael 793
di Fibonacci 33, 70, 802
naturali 73
primi 760, 791
primi e composti 761
reali 73
numero armonico 41
numero di parentesizzazioni 285

O

oggetti 197
allocazione e deallocazione 198
ombra 854
ombra di un punto 854
operazioni 423
a farfalla 750
che fondono heap 406
di giunzione (join) su RB-alberi 261
insiemi disgiunti 423
perno 714
su heap binomiali 389
sugli heap di Fibonacci 420
sulle matrici 692
operazioni su insiemi 74
intersezione 74
unione 74
operazioni su insiemi disgiunti 423
operazioni tra insiemi
differenza 74
opposto 770
OR 620
ordinamenti 13, 16
bitonico 608
bucket sort 170
counting sort 166
di segmenti 841
in loco in tempo lineare 173
insertion sort 2, 6, 16
limiti inferiori 163
lineari 78
merge sort 13, 16
modello ad albero di decisione 164
parziale 78
parziale dello heap 386
per confronti 163
quicksort 145, 147, 152, 154
radix sort 168
rete di ordinamento 613
topologico 465
totale 78
ordinatore bitonico 609
ordine di grandezza 9, 34
organizzazione hash con indirizzamento aperto 223
overflow 189

P

package wrapping 852
padre 87
pagine 364
palline e contenitori 120, 124
sequenze di successi 121
paradosso del compleanno 118
parametri
costo del passaggio dei parametri 69
per valore 5
parentesizzazioni, numero 285
parità 621
parsing 303
partizionamento dell'array 146
partizione 75, 77
partizione sul mediano-fra-tre 160
Pascal, triangolo di 97
passaggio dei parametri
costo 69
pattern 805, 819
permutazione 80, 94
dei bit rovesciati 356
di Giuseppe Flavio 276
reti di permutazione 616
perni 714
PERT 512
peso di cammino minimo 491
peso medio 521
peso medio minimo 521
pianificazione di una festa aziendale 308
pile 160, 189, 338, 342, 345, 379
memoria secondaria 379
overflow 189
profondità 160
underflow 189
polinomi 29, 34, 735, 737
addizione 735
coefficienti 735
comportamento asintotico 34
grado-limite 735
moltiplicazione 736
moltiplicazione veloce 740
rappresentazione a coefficienti 737
rappresentazione a punti 737
regola di Horner 737
valutazione 737
Pollard 798
porta logica 620
potenza non banale 765
potenze di un elemento 781
potenze esatte 62
pozzo 552
PRAM 651
salto dei puntatori 654
sincronizzazione e controllo 653
PRAM CRCW 653
PRAM EREW 653
predecessore 493
albero binario di ricerca 233
prefissi su una lista 658
prefisso 806
prestazioni del quicksort 147
previsione del riporto 626
Prim 482
primalità 792
primalità randomizzata di Miller-Rabin 793
primi tra loro a coppie 764
principio di inclusione ed esclusione 76
principio zero-uno 605
probabilità 99
assiomi 98
calcolo 98
condizionata 100
condizionata ed indipendenza 100
contatore probabilistico 125
distribuzione 98
distribuzione binomiale 107
distribuzione continua uniforme 99
distribuzione discreta 99
distribuzione geometrica 107
palline e contenitori 120
problema dell'assunzione 125
valore atteso di una variabile casuale 104
variabili casuali discrete 103
varianza e scarto quadratico medio 106
problema di cammini minimi 491
destinazione singola 492
sorgente singola 492
tra tutte le coppie 492
tra una coppia 492
problemi
abbinamento massimo in un grafo bip 570
alberi ammortizzati con peso bilanciato 358
alberi binari di ricerca con chi. i uguali 243
alberi di copertura minimo pe. rafi sparsi 488
alberi lessicografici 243
albero di copertura minimo 4
albero di copertura minimo c e un semianello chiuso 549
algoritmo del MCD binario
algoritmo di Karp per il ciclo peso medio minimo 521
algoritmo di partizionamento Lomuto 158
algoritmo di Shamir per la m plicazione di matrici booleane 730
algoritmo di Tarjan per i mina antenati comuni 442

algoritmo incrementale di Gabow per cammini minimi 520
algoritmo per il minimo albero di copertura con gli heap aggregabili 400
algoritmo per trovare il massimo efficiente rispetto al numero di processori 685
algoritmo Stooge Sort 159
ammissibilità 514
analisi del programma max 124
analisi delle operazioni sui bit dell'algoritmo di Euclide 802
analisi di heap d-ari 144
approssimazione della dimensione di una cricca massima 925
arbitraggio 520
assunzione 125
astratti 864
calcolabile in tempo polinomiale 866
calcolo dei ranghi 655
calcolo segmentato parallelo dei prefissi 685
cammini minimi 491
cammini minimi in grafi e-densi 549
chiusura transitiva di un grafo dinamico 549
cicli di Eulero 475
ciclo hamiltoniano 871, 897
ciclo semplice più lungo 903
circuiti di divisione 647
classificazione degli archi con visita in ampiezza 474
collocazione dell'ufficio postale 183
colorazione di un grafo 904
colorazione di un grafo di intervalli 315
commesso viaggiatore 901, 911, 912, 914
commesso viaggiatore bitonico e euclideo 306
componenti connesse 686
computazionali 1
concreti 866
confronti tra liste 203
contatore binario dei bit rovesciati 356
contatore probabilistico 125
continuo dello zaino 317
copertura di un insieme 916
copertura di vertici 892, 909
copertura pesata di un insieme 925
correttezza della procedura Partition 158
corrispondenza tra stringhe basata sui fattori di ripetizione 832
costo del pasaggio dei parametri 69
costruzione di uno heap con l'inserzione 143
cricca 890
determinazione della profondità 441
di decisione 865
di ottimizzazione 865
di ricerca 5
discreto dello zaino 317
distanza di editing 307
distribuzioni inviluppate sparse 860
esempi di ricorrenze 68
esperimenti con navette spaziali 594
estensione dalle potenze di un intero a tutti i reali 70
fantasmi ed acchiappafantasmi 859
FFT con aritmetica modulare 756
flusso massimo 553
formule Booleane per funzioni simmetriche 648
fuga 593
generale del commesso viaggiatore 914
hashing k-universale 228
heap 2-3-4 399
incapsulamento di scatole 519
insieme indipendente 903
insiemi dinamici persistenti 260
inviluppo convesso in tempo reale 854
istanza di un problema 2
limite alla dimensione della tabella nella concatenazione 227
limite alla scansione più lunga in un'organizzazione hash 226
limiti inferiori sull'ordinamento per confronti nel caso medio 173
matrici di Toeplitz 754
mediano pesato 182
miglioramento di Yen all'algoritmo di Bellman-Ford 519
minimo albero di copertura 328
minimo off-line 440
moltiplicazione divide-et-impera 754
NP-completi 863, 890, 891
numeri di Fibonacci 70
operazione di giunzione (join) su RB-alberi 261
operazioni sugli heap di Fibonacci 420
ordinamento 1
ordinamento in loco in tempo lineare 173
ordinamento su un unico processore di programmi di durata unitaria con scadenze e penalità 332
ottimizzazione 283, 865
palline e contenitori 124
partizione di un insieme 903
permutazione di Giuseppe Flavio 276
pianificazione di una festa aziendale 308
profondità della pila per il quicksort 160
profondità media di un nodo in un albero binario di ricerca costruito in modo casuale 243
programmazione intera 0-1 903

programmazione lineare 513
punti di articolazione, ponti e componenti biconnesse 474
punto di massima sovrapposizione 276
realizzazione alternativa della eliminazione 420
realizzazione di un mergeable heap con liste concatenate 204
residui quadratici 802
resto con monete 334
reti di fusione pari-dispari di Batcher 616
reti di ordinamento per trasposizione 615
reti di permutazione 616
ricerca 5
ricerca binaria dinamica 357
ricerca in un insieme statico 227
ricerca in una lista ordinata compatta 204
riempimento di contenitori 925
risolvibile in tempo polinomiale 866
scansione quadratica 227
scheduling 332, 335
selezione di attività 312
selezione di elementi piccoli 183
sequenza degli i numeri più grandi 182
sistemi tridiagonali di equazioni lineari 731
soddisfattibilità di circuiti 878
somma di sottoinsieme 894, 920
sottografi aciclici 335
sottografo isomorfo 903
sottosequenza comune più lunga 296
spline 731
stampa precisa 306
strati convessi 859
strati massimali 859
traslazione di un'immagine raster 687
tre algoritmi per i numeri di Fibonacci 802
triangolazione ottima (di un poligono) 302
trovare l'intero mancante 69
valutazione di polinomi in più punti 755
valutazione di tutte le derivate di un polinomio in un punto 755
procedure per foreste di insiemi disgiunti 430
prodotto 93, 736
cartesiano 76
delle sottomatrici 699
di una sequenza di matrici 284
esterno 693
incrociato 836, 838
interno 693
parziale 635
per uno scalare 692
scalare di flussi 558
prodotto di matrici
calcolo dei costi ottimi 287
completamente parentesizzato 284
costruzione di una soluzione ottima 290
determinazione del numero di parentesizzazioni 285
soluzione ricorsiva 286
struttura della parentesizzazione ottima 286
produttorie 42
profondità 623
dei circuiti 623
della pila 160
media di un nodo 243
programma di durata unitaria 332
programma max 124
programmazione dinamica 291, 292
analisi comparata degli algoritmi greedy e della programmazione dinamica 316
prodotto di una sequenza di matrici 284
ricorsione con memorizzazione 294
sottoproblemi comuni 292
sottostruttura ottima 291
triangolazione ottima di un poligono 302
programmazione lineare 513
propagazione del riporto 625
proprietà
degli alberi liberi 85
dei ranghi 435
dei segmenti 835
della notazione asintotica 35
della scelta greedy 316
delle matrici 689
di alberi binomiali 385
di ordinamento parziale dello heap 134
di scambio 326
RB 247
prospetto dei punti di arresto 842
prova di Bernoulli 107
pseudocodice 2, 4
pseudoinversa 730
pseudoprimalità 792
puntatori 197, 654
puntatori e oggetti 197
punti di articolazione, ponti e componenti biconnesse 474
punto di massima sovrapposizione 276

Q

quadratura ripetuta 531, 784
quantili 181
quasianello 704
quasianello booleano 704
quicksort 145
analisi 154
analisi del caso medio 154

analisi del caso peggiore 154
analisi del partizionamento 155
caso medio 150
limite inferiore per il caso peggiore 164
partizionamento bilanciato 149
partizionamento migliore 149
partizionamento peggiore 148
partizione sul mediano-fra-tre 160
prestazioni 147
profondità della pila 160
ricorrenza 156
ricorrenza per il caso medio 155
versione randomizzata 152
quoziente 761

R

Rabin-Karp 809
radici 87
radice n-esima complessa dell'unità 742
radice primitiva 782
radice primitiva n-esima dell'unità 742
radice quadrata non banale di 1, modulo n 783
radici complesse dell'unità 742, 747
radix sort 168
raggio orizzontale destro 840
RAM 6
random-access machine 6
ranghi 435, 655
rango 80, 155, 263, 429, 694
di colonna 694
di colonna pieno 694
di riga 694
di un elemento 265
pieno 694
rapporto limite 907
rappresentazione
a coefficienti 737
a lista concatenata di insiemi disgiunti 426
a punti 737
dei cammini minimi 493
di alberi radicati 201
di grafi 447
di oggetti con array multipli 197
di oggetti con un solo array 197
di polinomi 737
figlio-sinistro fratello-destro 202
rastrellamento 840, 842
RB 247
RB-alberi 247
cancellazione 255
estensione 269
inserzione 251
operazione di giunzione (join) 261
rotazioni 249
realizzazione alternativa della eliminazione 420
realizzazione iterativa della FFT 749
reciproco 707, 778
registri 641
regola di Horner 737
regole di somma e prodotto 93
relazione
di equivalenza 77
relazione binaria 77
antisimmetrica 78
riflessiva 77
simmetrica 77
transitiva 77
residui quadratici 802
resto con monete 334
rete
ammissibile 584
con sorgenti e pezzi multipli 555
di confrontatori 601
di permutazione 616
residua 559
rete di fusione 611
pari-dispari di Batcher 616
rete di ordinamento 604, 613
pari-dispari 615
per trasposizione 615
rete di ordinamento bitonico 608
half-cleaner 608
ordinatore bitonico 609
reti di flusso 552, 581
aggiornamento del flusso massimo 594
antisimmetria 552
calcolo incrementale del flusso massimo 594
cammini aumentanti 561
conservazione del flusso 552
esempio 553
flusso massimo con capacità superiori ed inferiori 595
per merci multiple 558
reti con sorgenti e pezzi multipli 555
reti residue 559
tagli 562
vincolo di capacità 552
retta di rastrellamento 840, 842
rettangolo delimitante 837
rho di Pollard 798
ricerca 204, 231
binaria dinamica 357
della chiave minima 389
della coppia di punti più vicini 854

di un abbinamento massimo bipartito 571
di un elemento con un dato rango 264
in un insieme statico 226
in una lista concatenata 193
in una lista ordinata compatta 204
lineare 5
lista concatenata 193
lista ordinata compatta 204
su B-alberi 369
ricorrenze 51, 59, 156
controllo di chip VLSI 71
esempi 68, 69
metodo di sostituzione 51, 53
metodo iterativo 51, 62
metodo principale 51, 62
soluzione candidata 59
sostituzione di variabili 62
teorema principale 62
ricorsione 62
alberi 62
con memorizzazione 294
rideterminazione dei pesi 539
riducibile in tempo polinomiale 875
riducibilità 875
rifiuto immediato 837
riflessività 27
rilassamento 495, 496, 497
riporto 625, 626, 628, 631
addizionatore senza riporto 631
di uscita 625
risoluzione
delle collisioni per concatenazione 210
deterministica dei conflitti 679
di sistemi di vincoli di differenza 517
risolvibile in tempo polinomiale 866
ritardo di propagazione 621
rotazioni 249
RSA 785, 788

S

salto dei puntatori 654
scansione lineare 221
scansione quadratica 221–227
scaricamento di un vertice traboccante 585
scarto quadratico medio 106
scatole KPG 631
schedule 332
scheduling 332
schema di approssimazione 908
pienamente polinomiale 921
setaccio 921
Schur 714
scomposizione di interi in fattori primi 797
scomposizione in fattori primi 764
scrittura-concorrente 652
scrittura-esclusiva 652
scritture concorrenti 665
segmenti 836, 841
confrontabili 841
consecutivi 837
ordinamento 841
orientati 836
selezione 175
con tempo medio lineare 177
di attività 312
di elementi piccoli 183
in tempo lineare nel caso peggiore 179
su un insieme dinamico 263
semianelli chiusi 543, 546, 549
sentinelle 194
sequenza
bitonica 608
degli i numeri più grandi 182
di esecuzione 332
di scansione 219
finita 79
infinita 79
serie
aritmetica 40
armonica 41
assolutamente convergente 40
di potenze formali 70
esponenziale 41
geometrica 40
telescopiche 41
setaccio 921
simbolo di Legendre 803
simmetria 27
simmetria trasposta 27
simplesso 513
sincronizzazione 653
sincronizzazione e controllo 653
singoletto 75
sistema di crittografia RSA 788
sistemi di equazioni lineari 708
sottodeterminato 709
sovradeterminato 709
sistemi di vincoli di differenza 514, 517
sistemi tridiagonali di equazioni lineari 731
slot 207
soddisfattibilità di circuiti 877
soddisfattibilità di formule 884
soluzione ai minimi quadrati 727
somma 93
Cartesiana 742
di flussi 558

sommatoria 39, 43, 45, 49, 157
sommatoria implicita 556
sorgente 450, 552
sostituzione 53
  a ritroso 711
  in avanti 710
sottoalbero
  mantenimento della dimensione 266
  radicato in x 87
sottocammini di cammini minimi 495
sottocammino 82
sottoespressione comune 749
sottografi aciclici 335
sottografo dei predecessori 456, 493, 526
sottogruppi 773, 774
sottogruppi generati da un elemento 774
sottoinsieme proprio 73
sottosequenza 296
sottosequenza comune 297
sottostringa 94
sottostruttura 897
  di una triangolazione ottima 305
  ottima 316
  ottima di un cammino minimo 495
sottostruttura ottima di una LCS 297
  calcolo della lunghezza di una LCS 298
  costruzione di una LCS 299
sottrazione tra matrici 692
spazio campione 98
speranza matematica 104
spline 731
spline cubiche 731
spostamento 805
stampa precisa 306
stati del riporto 627
statistica d'ordine 175
stella di Kleene 868
stima di cammino minimo 496
Stooge Sort 159
Strassen 697
strategia greedy 315
strati convessi 859
strati massimali 859
stringhe 94, 805, 814
  corrispondenza tra stringhe con gli automi a stati 813
  stringa vuota 806, 868
struttura
  della parentesizzazione ottima 286
  di un cammino minimo 527
  ricorsiva 10
strutture di dati 268, 364
  insiemi disgiunti 424
successore
  albero binario di ricerca 233
  e predecessore 233
suffisso 806
superpozzo 555
supersorgente 555
surgettiva 80
sviluppo binomiale 95

T

tabella di verità 620
tabelle
  hash 209, 211
  indirizzamento diretto 207
tabelle dinamiche 348
  espansione della tabella 349
  espansione e contrazione di tabelle 352
tabelle hash 209
  analisi dell'organizzazione hash con indirizzamento aperto 223
  hashing k-universale 228
  interpretazione delle chiavi come numeri naturali 214
  limite alla dimensione della tabella nella concatenazione 227
  limite alla scansione più lunga 226
  ricerca in un insieme statico 226
  scansione quadratica 227
tagli in reti di flusso 562
taglio 562
taglio a cascata 415
Tarjan 442
tasso di crescita 9
tasso di errore 795
tautologia 874
tempo di accesso 364
tempo di esecuzione 6, 9, 16, 530
  caso peggiore 8
  confronti 16
  dell'algoritmo di Euclide 767
  effetto delle euristiche 431
  notazione Ω 24
  notazione asintotica 21
  notazione O 23
  notazione o 26
  notazione Q 22
  notazione ω 27
tempo polinomiale 760, 864, 866, 868
teorema principale 59
  albero di ricorsione 62
  base 66
  dimostrazione 62
  dimostrazione per potenze esatte 62
  tetto 66

teoremi 101
  d'un vertice traboccante può essere applicata un'operazione di base 578
  annidamento degli intervalli dei discendenti 462
  Bayes 101
  cinese del resto 779
  correttezza della visita in ampiezza 455
  correttezza dell'algoritmo di Bellman-Ford 509
  correttezza dell'algoritmo di Dijkstra 503
  correttezza dell'algoritmo di Graham 850
  correttezza dell'algoritmo generico di preflusso 579
  correttezza dell'algoritmo greedy sui matroidi 331
  correttezza di RSA 789
  estensione di un RB-albero 269
  dei numeri primi 791
  del cammino bianco 462
  della divisione 761
  delle parentesi 461
  di Brent 670
  di convoluzione 748
  di divisione, dei resti e di congruenza 761
  di Eulero 782
  di Fermat 782
  di Hall 574
  di integrità 573
  di Lagrange 774
  di Lamé 768
  di ricorsione del MCD 766
  diseguaglianza della funzione suffisso 816
  flusso massimo taglio minimo 564
  i matroidi soddisfano la proprietà della sottostruttura ottima 330
  la rete ammissibile è aciclica 584
  le altezze dei vertici non decrescono mai 578
  lemma del complemento di Schur 725
  lemma della sommatoria 744
  lemma delle strette di mano 84
  lemma dell'iterazione della funzione prefisso 823
  lemma di cancellazione 742
  lemma di dimezzamento, 743
  lemma di ricorsione della funzione suffisso 817
  lemma di sovrapposizione dei suffissi 806
  limite superiore al numero di invii saturanti e non saturanti 581
  limite superiore alle operazioni di innalzamento 581
  l'inversione non è più gravosa della moltiplicazione 722
  logaritmo discreto 783
  matrici su un anello formano un anello 705
  matrici su un quasianello formano un quasianello 705
  principale 62
  principio zero-uno 607
  proprietà degli alberi liberi 85
  proprietà del determinante 695
  proprietà di alberi binomiali 39
  rideterminazione dei pesi non cambia i cammini minimi 539
  sottocammini di cammini minimi sono cammini minimi 495
  sottostruttura ottima di una LCS 297
  un sottoinsieme chiuso di un gruppo finito è un sottogruppo 773
  unicità del polinomio di interpolazione 738
  unicità della scomposizione in fattori primi 764
teoria dei numeri 760
termine essenziale 701
test di primalità di Miller-Rabin 795
testa 190
testo cifrato 787
tetto 29, 66
Toeplitz 754
traccia 364
transitività 27
trasformata discreta di Fourier 745
  rilassata 749
  veloce di Fourier 745
traslatore ciclico 641
traslazione di un'immagine raster 687
trasposizione
  reti di ordinamento 615
trasposta 449, 690
trasposta coniugata 724
triangolazione 302
triangolazione ottima di un poligono 302
  soluzione ricorsiva 305
  sottostruttura di una triangolazione ottima 305
triangolo di Pascal 97
tricotomia 28
tricotomia degli intervalli 271

U

underflow 189
unicità del polinomio di interpolazione 738
uniformità della funzione hash 220
uniformità semplice della funzione hash 211
unione 74, 868
  di due heap binomiali 390
  e divisione di alberi 2-3-4 380
  per rango 429
  pesata 426

unità di blocco 437
unità di cammino 437
unità di mezzo 437
uno-a-uno 80

V

valore 7
atteso di una variabile casuale 104
hash 209
valutazione
del polinomio 737
di polinomi in più punti 755
di tutte le derivate di un polinomio in un punto 755
Vandermonde 697
variabili casuali (discrete) 104
varianza 106
varianza e scarto quadratico medio 106
verifica
dell'intersezione di una coppia qualsiasi 840
dell'intersezione tra due segmenti 837
di primalità randomizzata di Miller-Rabin 793
vertice 81
isolato 82
traboccante 585
vettore 690, 836
collineare 836
colonna 690
di bit 208
riga 690
unità 690
vettori linearmente dipendenti 694
vettori linearmente indipendenti 694
vincoli 515
di capacità 552
di differenza 513, 514, 517
di differenza e cammini minimi 513
visita
completa 913
in ampiezza 450
in ordine anticipato 230
in ordine differito 230
in ordine simmetrico 230
in profondità 458
Viterbi
algoritmo 308

W

Wallace, moltiplicatori ad albero 638

X

XOR 620

Y

Yen, miglioramento all'algoritmo di Bellman-Ford 519

Z

zero-uno 605

INV. 245

UNIVERSITA' DI NAPOLI FEDERICO
BIBLIOTECA CENTRALE
FACOLTA' DI SCIENZE M.F.N.